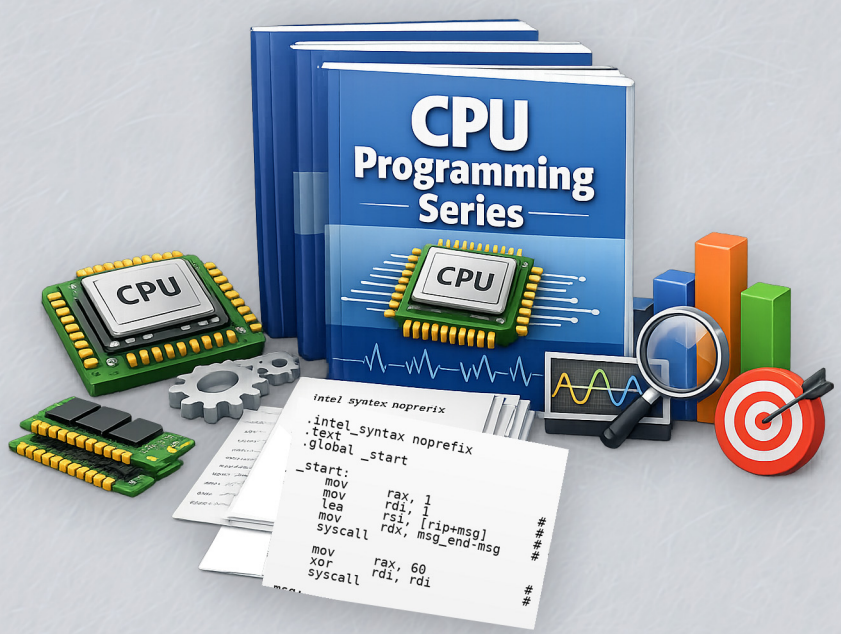
https://simplifycpp.org

# CPU Programming Series

## AArch64 Calling Convention (AAPCS64)

### C/C++ Interoperability in Practice




13

Prepared by Ayman Alheraki

# CPU Programming Series

## AArch64 Calling Convention (AAPCS64)

C/C++ Interoperability in Practice

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

# Contents

# Preface

## Purpose of This Booklet

This booklet teaches the **AArch64 Procedure Call Standard (AAPCS64)** as a **practical engineering discipline** for building reliable boundaries between:

- **C/C++ code compiled by different compilers/flags**,
- **handwritten AArch64 assembly**,
- and **independent binary modules** (static or dynamic) that must interoperate without undefined behavior.

The goal is not memorizing register tables; it is mastering the **invariants** that keep programs correct under optimization:

- how arguments are passed and returned,
- which registers must survive a call,
- what stack alignment and frame rules must be preserved,
- and where ABI violations become **silent data corruption**.

## What You Will Be Able to Do After This Booklet

- Write ABI-correct AArch64 assembly functions callable from C/C++.
- Call C/C++ functions from assembly without breaking the compiler's expectations.
- Diagnose wrong results caused by stack misalignment or register preservation bugs.
- Understand compiler-generated prologues/epilogues and what they guarantee.

## Minimal "Hello ABI" Example

The simplest correct boundary is a leaf function that:

- uses only caller-saved registers, and
- does not break stack alignment (and ideally does not touch SP at all).

```cpp
/* C/C++ side */
#include <cstdint>

extern "C" std::uint64_t add8_u64(std::uint64_t x);

std::uint64_t demo(std::uint64_t a) {
    return add8_u64(a); /* must follow AAPCS64 */
}
```

```asm
/* AArch64 GAS side (AAPCS64-compliant leaf) */
.text
.global add8_u64
.type   add8_u64, %function
add8_u64:
    /* x0 holds the first integer argument and return value */
    add     x0, x0, #8
    ret
```

# Where AAPCS64 Fits in the CPU Programming Series

AAPCS64 is the **bridge** between:

- **instruction-level execution** (registers, flags, addressing),
- and **system-level software** (compilers, linkers, OS interfaces, libraries).

In this series, the progression is intentional:

- Earlier booklets explain **how CPUs execute instructions** and how registers/memory behave.
- The ABI booklets establish **stack and calling discipline** as a cross-language contract.
- This booklet focuses on the **AArch64 user-space ABI contract** used by real C/C++ programs on modern ARM64 platforms.

## Why This Booklet Exists (Even If You "Know Assembly")

Knowing instructions is not enough. Most real bugs at the boundary are not "wrong instruction" bugs; they are **contract violation** bugs:

- clobbering a callee-saved register used by the caller,
- misaligning SP (breaking ABI-required alignment),
- returning a value in the wrong register,
- or assuming a layout for aggregates that the ABI defines differently.

### A Quick Map to Neighbor Booklets

- If you struggle with **stack frames**, revisit the booklet on stack and calling conventions (architecture-neutral foundations).
- If you struggle with **register meaning**, revisit the AArch64 core architecture booklet (register roles, PSTATE basics, addressing).
- If you need **syscall boundary rules**, use the syscall/privilege-boundary booklet (user vs kernel entry discipline).

## Prerequisites and Assumed Knowledge

This booklet assumes you can read short AArch64 code sequences and recognize:

- general-purpose registers `x0--x30`, `sp`, and `lr` (`x30`),
- basic load/store and branch instructions,
- and the idea of a function call changing control flow while preserving a caller-visible state.

### Required C/C++ Concepts

- Function signatures, return types, and parameter passing.
- The meaning of `extern "C"` as a linkage contract (name mangling boundary).
- Basic understanding that optimization may reorder/register-allocate aggressively, *as long as ABI rules are respected*.

### Tooling Assumptions (Minimal)

- You can build and link C/C++ and assembly together (any modern toolchain).
- You can inspect symbols and disassembly when debugging ABI issues.

## Scope, Limits, and ABI Discipline Philosophy

### Scope

This booklet covers the **AAPCS64 calling convention** needed for **C/C++ interoperability in practice**:

- integer/pointer argument and return rules,
- floating-point/SIMD argument and return rules (at the ABI contract level),
- caller-saved vs callee-saved register preservation,
- stack alignment and safe frame construction,
- and the minimum rules for mixing compiler code with handwritten assembly.

### Out of Scope (By Design)

To keep this booklet focused and reliable, the following are treated only conceptually or deferred:

- C++ exception unwinding internals and personality routines (toolchain-specific).
- OS kernel ABIs and syscall conventions (separate booklet).
- Microarchitecture performance tuning (separate performance/caches booklets).
- Full DWARF unwind metadata authoring (advanced debugging topic).

## ABI Discipline Philosophy: "Correct First, Fast Second"

At ABI boundaries, **correctness is a property of invariants**. Performance only matters after invariants hold. Follow this discipline:

- **Preserve what must be preserved** (callee-saved registers).
- **Align what must be aligned** (stack pointer rules).
- **Return where the ABI expects** (return registers and conventions).
- **Never guess layouts** (especially for aggregates); use ABI rules or let the compiler do it.

## A "Wrong but Looks Fine" Example (Register Preservation Bug)

The following assembly violates ABI discipline by clobbering a register that the caller expects to survive across the call. It may work in a small test and fail under optimization or in a larger program.

```
/* BAD: clobbers a callee-saved register (example uses x19 as a
 ↪  preserved register) */
.text
.global bad_clobber
.type   bad_clobber, %function
bad_clobber:
    /* x19 is commonly treated as callee-saved by the ABI contract */
    mov     x19, #1234        /* ABI violation: overwrites caller's
     ↪  state */
    add     x0, x0, #1
    ret
```

Correct approach: if you use a callee-saved register, **save/restore it** and keep stack alignment intact.

```
/* GOOD: saves/restores preserved state and keeps stack aligned */
.text
.global good_preserve
.type   good_preserve, %function
good_preserve:
    /* Maintain 16-byte alignment when adjusting SP */
    sub     sp, sp, #16
    str     x19, [sp, #0]       /* save callee-saved register */
    /* body */
    mov     x19, #1234          /* now allowed */
    add     x0, x0, #1
    /* restore */
    ldr     x19, [sp, #0]
    add     sp, sp, #16
    ret
```

## A Minimal Non-Leaf Template (Calls Another Function)

Any function that calls another function must assume caller-saved registers can be destroyed by the callee. A safe minimal template saves what it must keep, and keeps SP aligned.

```
/* Non-leaf example: calls an external C function: uint64_t
 ↪  g(uint64_t); */
.text
.global f_calls_g
.type   f_calls_g, %function
f_calls_g:
    /* Save LR if you need it after the call; keep stack aligned */
    sub     sp, sp, #16
    str     x30, [sp, #8]       /* save return address (LR) */
```

```
/* x0 already contains the first argument; call g(x0) */
bl      g                   /* may clobber caller-saved regs */

/* Restore LR and return */
ldr     x30, [sp, #8]
add     sp, sp, #16
ret
```

# Chapter 1

# AAPCS64 Overview and Design Goals

## 1.1 What a Calling Convention Really Defines

A calling convention is a **binary contract** that allows separately-compiled code to call each other correctly. It defines **what must be true at every call boundary** so that:

- a caller can pass arguments and receive results,
- a callee can freely use some CPU state while preserving required state,
- and both sides remain correct under compiler optimization and across translation units.

### 1.1.1 The Four Things Every Calling Convention Must Define

1. **Argument passing:** where each parameter goes (registers vs stack) and in what order.
2. **Return values:** where results are placed (which register(s), or via hidden pointers).
3. **Register volatility:** which registers a callee may destroy (caller-saved) and which it must preserve (callee-saved).

4. **Stack discipline:** stack direction, alignment rules, and what the callee may assume about `SP`.

### 1.1.2 What It Does *Not* Define

A calling convention does *not* define the semantics of the language, algorithms, or the optimizer. It only defines the **interface invariants**. For example, it does not require a specific prologue/epilogue sequence; it requires that the **observable contract is satisfied** (preservation, alignment, and correct parameter/result mapping).

### 1.1.3 Interoperability Is "State Management"

When C/C++ interoperate with assembly, your real job is to manage **state across the boundary**:

- preserve what must be preserved,
- align what must be aligned,
- and avoid assuming layouts or side effects not guaranteed by the ABI.

### 1.1.4 Example 1: One-Argument, One-Return (Register Contract)

In AAPCS64, the first integer/pointer argument is passed in `x0`, and the integer/pointer return value is placed in `x0`. This makes simple leaf functions extremely efficient and safe.

```
/* C/C++ caller */
#include <cstdint>

extern "C" std::uint64_t inc1(std::uint64_t x);

std::uint64_t demo(std::uint64_t a) {
```

```
    return inc1(a);
}
```

```
/* AArch64 GAS callee: x0 = arg0, x0 = return */
.text
.global inc1
.type   inc1, %function
inc1:
    add     x0, x0, #1
    ret
```

### 1.1.5 Example 2: Caller-Saved Reality (Why Values "Disappear")

Caller-saved registers may be clobbered by a call. If you keep a live value only in caller-saved registers and then call another function, you must assume it can be destroyed.

```
/* Concept: preserve a value across a call using the stack */
.text
.global keep_value_across_call
.type   keep_value_across_call, %function
keep_value_across_call:
    /* Suppose x0 has some important value we need after calling g */
    sub     sp, sp, #16
    str     x0, [sp, #0]      /* save live value */

    bl      g                 /* g may clobber caller-saved regs */

    ldr     x0, [sp, #0]      /* restore the saved value */
    add     sp, sp, #16
    ret
```

# 1.2 AAPCS64 in the ARM Architecture Ecosystem

## 1.2.1 ISA vs ABI: Two Different Documents

**AArch64** is the instruction set architecture (ISA): it defines registers, instructions, exception levels, and the architectural memory model.
**AAPCS64** is the procedure call standard (ABI contract): it defines **how software uses that ISA at call boundaries**. Two compilers can generate different instruction sequences, but both must satisfy the same ABI rules to interoperate.

## 1.2.2 Ecosystem Layers (Where AAPCS64 Sits)

Think in layers:

- **ISA layer:** AArch64 instructions, registers, exception levels.
- **ABI layer:** AAPCS64 calling convention, stack alignment rules, register preservation rules.
- **Platform ABI layer:** OS runtime choices, object format (ELF/Mach-O/PE), dynamic linker conventions.
- **Language ABI layer:** C++ name mangling, exception handling model, RTTI conventions (toolchain/platform-specific).

This booklet is about **AAPCS64 as the interop contract for C/C++ and assembly**. Platform-specific C++ ABI details are referenced only conceptually and are handled in advanced booklets.

### 1.2.3 Why AAPCS64 Enables "Mix-and-Match"

When a C function compiled by one toolchain can call an assembly function written by you, it is because both follow:

- the same register argument/return mapping,
- the same definition of preserved registers,
- and the same stack alignment invariants.

### 1.2.4 Example: Mixed Integer and FP Arguments

AAPCS64 splits integer/pointer and floating-point argument passing into separate register banks (GPRs and SIMD/FP registers). The contract is what makes mixed signatures callable without ambiguity.

```
/* C/C++ signature with mixed arguments */
extern "C" double mix(std::uint64_t a, double x, std::uint64_t b, double
↪  y);
```

```
/* AArch64 GAS sketch (concept-only mapping) */
.text
.global mix
.type   mix, %function
mix:
    /* a in x0, x in v0 (d0), b in x1, y in v1 (d1) */
    /* body omitted: the point is distinct register banks */
    fadd    d0, d0, d1          /* return double in d0 */
    ret
```

# 1.3 ABI Stability and Long-Term Compatibility

## 1.3.1 Why ABI Stability Matters

Large systems survive for decades because ABI contracts remain stable. ABI stability allows:

- updating a shared library without recompiling every consumer,
- linking modules built by different teams at different times,
- and shipping prebuilt binaries that remain interoperable across toolchain upgrades.

A stable ABI is more valuable than any single optimizer trick because it protects **independent evolution** of components.

## 1.3.2 What Makes an ABI "Stable"

An ABI is stable when its observable rules do not change:

- which registers carry which arguments/returns,
- which registers must be preserved,
- the required stack alignment at public interfaces,
- and the canonical rules for aggregates and variadic calls.

The compiler may change instruction selection, scheduling, and register allocation freely as long as these invariants remain true.

### 1.3.3 Long-Term Compatibility Strategy

For long-lived software, treat ABI as a **public boundary**:

- Keep exported interfaces small and explicit.
- Prefer C-compatible boundaries for stable plugin/FFI interfaces.
- Avoid exposing C++ types (templates, exceptions, STL) across module boundaries unless the entire toolchain and platform ABI are controlled.

### 1.3.4 Example: Stable C ABI Wrapper Around C++

This pattern keeps the external ABI stable even if internal C++ implementation changes.

```
/* Public ABI: stable C boundary */
extern "C" void* create_obj();
extern "C" void  destroy_obj(void* p);
extern "C" std::uint64_t compute_obj(void* p, std::uint64_t x);
```

```
/* Internal: C++ implementation can evolve */
#include <cstdint>
#include <new>

struct Obj {
    std::uint64_t bias = 7;
    std::uint64_t compute(std::uint64_t x) const { return x + bias; }
};

extern "C" void* create_obj() {
    return new (std::nothrow) Obj{};
}

extern "C" void destroy_obj(void* p) {
```

```
    delete static_cast<Obj*>(p);
}

extern "C" std::uint64_t compute_obj(void* p, std::uint64_t x) {
    return static_cast<Obj*>(p)->compute(x);
}
```

# 1.4 User Space vs Kernel Space Conventions (High-Level)

## 1.4.1 Two Worlds, Two Contracts

AAPCS64 defines the user-space procedure call standard for typical application code. Kernel entry/exit is a different boundary:

- user-space function calls preserve an ABI-defined contract between two pieces of **user code**,
- kernel transitions preserve an OS-defined contract between **user mode and privileged mode**.

These contracts overlap in the sense that both must preserve architectural correctness, but their goals differ.

## 1.4.2 High-Level Differences

- **Privilege boundary:** kernel entry changes exception level and uses privileged state.
- **Entry mechanism:** kernel entry is typically via a defined instruction or exception path, not a normal `bl`.
- **Register meaning:** an OS syscall ABI may reuse registers for syscall numbers and arguments; it is not automatically “the same as AAPCS64 calls”.

- **State saving:** kernel must save/restore enough state to safely return to user code, often beyond what a normal function call preserves.

### 1.4.3 Why You Must Not Mix Them Mentally

A common error is to assume:

> "If I know AAPCS64, I know the syscall convention."

This is false. The syscall boundary is an OS-defined ABI layered on top of the architecture exception mechanism.

### 1.4.4 Conceptual Example: "Call" vs "Privilege Entry"

```
/* Normal call boundary: AAPCS64 rules */
.text
.global normal_call_example
.type   normal_call_example, %function
normal_call_example:
    /* x0 = arg0 */
    bl      some_function      /* normal function call */
    ret
```

```
/* Privilege boundary: conceptual syscall-style entry (OS-defined
 ↪  contract) */
.text
.global kernel_entry_concept
.type   kernel_entry_concept, %function
kernel_entry_concept:
    /* The OS may require a syscall number in a specific register and
     ↪  args in others */
```

```
    /* Entry instruction triggers an exception-level transition
     ↪  (concept only) */
    /* Do NOT treat this like a normal 'bl' call */
    /* placeholder: entry mechanism is OS-specific and handled in a
     ↪  dedicated booklet */
    ret
```

### 1.4.5 Booklet Separation Policy

This booklet treats kernel conventions only at a high level to prevent ABI confusion. The **complete syscall entry ABI**, register usage, and signal/exception interactions belong to the dedicated privilege-boundary/syscall booklet in this series.

# Chapter 2

# Fundamental Register Roles in AAPCS64

## 2.1 General-Purpose Registers (X0–X30): Conceptual Roles

AArch64 provides 31 general-purpose registers `x0--x30` (64-bit). Their **architectural meaning** is generic, but the ABI assigns **conventional roles** so independently built code can interoperate.

### 2.1.1 The ABI View: Registers as an Interface Contract

In AAPCS64, registers have **roles at call boundaries**:

- **Parameter/result registers:** where values enter and leave a function.
- **Temporaries:** scratch registers that a call may freely clobber.
- **Preserved registers:** registers whose values must survive a call.
- **Special-purpose:** link register and platform register conventions.

### 2.1.2 Canonical Concepts You Must Remember

- **`x0--x7`** are the primary integer/pointer **argument registers**.
- **`x0`** is also the primary integer/pointer **return register**.
- **`x30`** is the **link register (LR)** holding the return address after a call.
- **`x29`** is commonly used as a **frame pointer (FP)** when a frame is established.
- **Some registers are callee-saved:** if you use them, you must restore them before returning.

### 2.1.3 A Practical Mental Model

Treat the call boundary like a strict checkpoint:

- the caller sets up `x0--x7` (and stack overflow arguments),
- the callee consumes them and is free to overwrite volatile state,
- then the callee returns the result in `x0` (or via defined multi-register/indirect rules),
- while preserved registers remain unchanged from the caller's view.

## 2.2 Argument Registers vs Temporary Registers

### 2.2.1 Argument Registers Are Not "Yours" After a Call

A frequent source of bugs is assuming that argument registers keep their values across calls. Under AAPCS64:

- argument registers (e.g., `x0--x7`) are typically **caller-saved**,

- therefore a function call may overwrite them,
- so if a value in `x0` matters *after* a call, you must preserve it.

### 2.2.2 Example 1: Losing an Argument Across a Call (Bug Pattern)

```
/* BAD pattern: expects x0 to still hold the original value after
 ↪  calling helper */
.text
.global bad_use_after_call
.type   bad_use_after_call, %function
bad_use_after_call:
    /* x0 = input */
    bl      helper              /* helper may clobber x0..x7 */
    add     x0, x0, #1           /* BUG: x0 may no longer be the
     ↪  original input */
    ret
```

### 2.2.3 Example 2: Preserve the Value Across the Call (Correct)

```
/* GOOD: preserve x0 across the call using the stack (keeps SP
 ↪  aligned) */
.text
.global good_use_after_call
.type   good_use_after_call, %function
good_use_after_call:
    sub     sp, sp, #16
    str     x0, [sp, #0]         /* save original input */

    bl      helper               /* may clobber caller-saved regs */
```

```
    ldr     x0, [sp, #0]          /* restore original input */
    add     sp, sp, #16
    add     x0, x0, #1
    ret
```

### 2.2.4 When to Use a Preserved Register Instead of the Stack

If you already need a stack frame, saving values on the stack is straightforward and universal. If performance or structure suggests using a preserved register, you may do so *only if you follow callee-saved rules*.

## 2.3 Callee-Saved vs Caller-Saved Discipline

### 2.3.1 Definitions (ABI Contract)

**Caller-saved (volatile)** registers:

- may be overwritten by the callee,
- so the caller must save them if it needs their values after the call.

**Callee-saved (non-volatile)** registers:

- must be preserved by the callee,
- so if the callee uses them, it must save/restore them before returning.

This is not “style”. It is the **interoperability law** that makes separately-compiled code safe.

### 2.3.2 Why Compilers Rely on This

Optimizing compilers allocate long-lived variables into callee-saved registers specifically because they are guaranteed to survive calls. If you violate that, you cause:

- silent corruption,
- failures that appear only with optimization,
- and bugs that vanish under debugging (Heisenbugs).

### 2.3.3 Example 3: Clobbering a Preserved Register (Catastrophic Bug)

```
/* BAD: clobbers a callee-saved register (example uses x19 as
 ↪  preserved) */
.text
.global bad_clobber_x19
.type   bad_clobber_x19, %function
bad_clobber_x19:
    mov     x19, #0             /* ABI violation */
    add     x0, x0, #1
    ret
```

### 2.3.4 Example 4: Correct Use of a Callee-Saved Register

```
/* GOOD: save/restore x19, keep SP 16-byte aligned */
.text
.global good_use_x19
.type   good_use_x19, %function
good_use_x19:
    sub     sp, sp, #16
```

```
    str     x19, [sp, #0]          /* save preserved register */

    mov     x19, x0                /* use x19 as a long-lived local */
    add     x19, x19, #42
    mov     x0, x19                /* return in x0 */

    ldr     x19, [sp, #0]          /* restore preserved register */
    add     sp, sp, #16
    ret
```

### 2.3.5 A Reliable Rule for Handwritten Assembly

If you are unsure whether a register is preserved or volatile:

- treat it as **volatile** and save what you need,
- or use a strict function template that saves/restores any preserved register you touch,
- and never ship assembly that "seems to work" without being provably ABI-correct.

## 2.4 Zero Register (XZR) and Stack Pointer (SP) Constraints

### 2.4.1 XZR: Reads as Zero, Writes Discard

`XZR` is not a normal general-purpose register:

- reading `xzr` produces **0**,
- writing to `xzr` discards the result.

This is useful for common idioms:

- zeroing a register without loading an immediate,
- comparisons against zero,
- and forming instructions without dedicating a register to constant zero.

### 2.4.2 Example 5: Efficient Zeroing and Zero-Compare

```
/* x1 = 0, then test x0 == 0 without allocating a constant register
 ↪  */
.text
.global zero_idioms
.type   zero_idioms, %function
zero_idioms:
    mov     x1, xzr             /* x1 = 0 */
    cmp     x0, xzr             /* sets flags based on x0 - 0 */
    cset    x0, eq              /* x0 = 1 if x0 == 0 else 0 */
    ret
```

### 2.4.3 SP: Special Register With Hard ABI Constraints

`SP` is not a general register. The ABI requires strict rules because:

- the stack is shared infrastructure for calls, spills, locals, and unwinding,
- and many components (compiler, debugger, unwinder, sanitizer) assume valid stack discipline.

### 2.4.4 The Non-Negotiable SP Rules for This Booklet

- **SP must remain 16-byte aligned at public call boundaries.**

- Any SP adjustment must preserve that alignment.
- Stack allocation must be undone before `ret`.
- Do not treat `sp` like a scratch register.

### 2.4.5 Example 6: Correct Stack Allocation (Aligned)

```
/* Allocate 32 bytes (multiple of 16) for locals */
.text
.global aligned_stack_locals
.type   aligned_stack_locals, %function
aligned_stack_locals:
    sub     sp, sp, #32
    /* use [sp, #0..31] as local storage */
    add     x0, x0, #5
    add     sp, sp, #32
    ret
```

### 2.4.6 Example 7: Misalignment Bug (Often Silent, Always Dangerous)

```
/* BAD: breaks 16-byte alignment */
.text
.global misalign_sp_bug
.type   misalign_sp_bug, %function
misalign_sp_bug:
    sub     sp, sp, #8          /* ABI violation: SP no longer
     ↪  16-byte aligned */
    /* may appear to work until a callee assumes aligned stack for
     ↪  spills or SIMD */
    add     x0, x0, #1
```

```
    add     sp, sp, #8
    ret
```

### 2.4.7 Practical Guidance

- Always allocate stack in multiples of **16 bytes**.
- Prefer storing paired registers with aligned slots when building larger frames.
- If your function calls other functions, treat stack alignment as **a precondition you must maintain**.

# Chapter 3

# Floating-Point and SIMD Register Convention

## 3.1 V0–V7: Floating-Point Argument Passing

AAPCS64 defines a **separate register bank** for floating-point and SIMD arguments using the vector registers `v0--v31`. At the ABI boundary:

- floating-point scalar arguments (`float`, `double`) are passed in `v0--v7` (using the `s` or `d` view: `s0/d0`..),
- small vector/SIMD arguments may also be passed in `v0--v7` according to their ABI classification,
- floating-point return values are returned in `v0` (e.g., `s0` or `d0`).

This separation is essential: it allows a single function signature to carry both integer/pointer and FP/vector values efficiently without forcing everything onto the stack.

### 3.1.1 Example 1: Pure Floating-Point Call Boundary (double)

```
/* C/C++ */
extern "C" double add_d(double a, double b);

double demo(double x, double y) {
    return add_d(x, y);
}
```

```
/* AArch64 GAS: a in d0, b in d1, return in d0 */
.text
.global add_d
.type   add_d, %function
add_d:
    fadd    d0, d0, d1
    ret
```

### 3.1.2 Example 2: float Variant (s-register view)

```
/* C/C++ */
extern "C" float mul_f(float a, float b);

float demo_f(float x, float y) {
    return mul_f(x, y);
}
```

```
/* AArch64 GAS: a in s0, b in s1, return in s0 */
.text
.global mul_f
.type   mul_f, %function
mul_f:
    fmul    s0, s0, s1
    ret
```

### 3.1.3 Example 3: Overflow to Stack (Conceptual)

`v0--v7` provide the primary FP argument slots. If a function has more FP arguments than available registers, the ABI spills remaining arguments to the stack according to the ABI layout rules. This booklet treats overflow as a **conceptual rule** now; the exact stack layout mechanics are covered in the stack/argument chapters.

## 3.2 Callee-Saved SIMD Registers

### 3.2.1 Volatile vs Preserved in the SIMD Bank

Just as with general-purpose registers, AAPCS64 defines which SIMD registers are:

- **caller-saved (volatile):** may be clobbered by a call,
- **callee-saved (preserved):** must survive across a call.

This matters immediately when you write assembly that:

- uses vector registers for temporaries,
- calls other functions,
- or expects vector state to survive across calls.

### 3.2.2 Example 4: Volatile SIMD Register Clobber (Bug Pattern)

```
/* BAD: caller assumes v0 survives a call (it generally must not) */
.text
.global bad_assume_v0_survives
.type   bad_assume_v0_survives, %function
bad_assume_v0_survives:
```

```
    /* d0 holds a live value */
    bl      helper_fp           /* helper_fp may clobber v0..v7 and
     ↪  more */
    fadd    d0, d0, d0          /* BUG: d0 may no longer be the
     ↪  original value */
    ret
```

### 3.2.3 Example 5: Preserve a Live FP Value Across a Call (Spill to Stack)

A safe, universal method is to spill the live FP value to the stack before the call and restore it after. Keep `sp` aligned.

```
/* GOOD: preserves d0 across the call using stack spill */
.text
.global good_preserve_d0_across_call
.type   good_preserve_d0_across_call, %function
good_preserve_d0_across_call:
    sub     sp, sp, #16
    str     d0, [sp, #0]        /* save scalar FP value */

    bl      helper_fp

    ldr     d0, [sp, #0]        /* restore scalar FP value */
    add     sp, sp, #16
    fadd    d0, d0, d0
    ret
```

### 3.2.4 Example 6: Using a Preserved SIMD Register (Template)

If you choose to use a callee-saved SIMD register as a long-lived local, you must save/restore it. The safest form is storing it to the stack.

```
/* Template: save/restore a SIMD register before using it as a
 ↪  long-lived local */
.text
.global use_preserved_simd_template
.type   use_preserved_simd_template, %function
use_preserved_simd_template:
    sub     sp, sp, #32
    /* Save a 128-bit vector register (q-view) into aligned stack
     ↪  space */
    str     q16, [sp, #0]       /* example uses q16 as a preserved
     ↪  local */
    /* body */
    fmov    d0, #1.0            /* placeholder work */
    /* restore */
    ldr     q16, [sp, #0]
    add     sp, sp, #32
    ret
```

## 3.3 Mixed Integer and Floating-Point Arguments

### 3.3.1 Two Independent Allocation Streams

AAPCS64 allocates integer/pointer arguments primarily from `x0--x7` and FP/SIMD arguments primarily from `v0--v7`. These are **independent streams**:

- integer arguments do not consume FP slots,
- FP arguments do not consume integer slots,
- the mapping is determined by the **function signature and ABI classification**.

### 3.3.2 Example 7: Mixed Signature With Clear Mapping

```
/* C/C++ */
#include <cstdint>

extern "C" double mix2(std::uint64_t a, double x, std::uint64_t b, double
↪  y);

/* Conceptual ABI mapping:
   a -> x0
   x -> d0
   b -> x1
   y -> d1
   return -> d0
*/
```

```
/* AArch64 GAS: mixed argument bank usage */
.text
.global mix2
.type   mix2, %function
mix2:
    /* a in x0, b in x1, x in d0, y in d1 */
    /* Use integers without touching FP regs, and FP without touching
    ↪  integer regs */
    add     x0, x0, x1          /* example integer work (not returned
    ↪  here) */
```

```
    fadd    d0, d0, d1          /* return FP result in d0 */
    ret
```

### 3.3.3 Example 8: A Common Mistake (Wrong Bank Assumption)

```
/* BAD: assumes the second parameter (double) is in x1; it is in d0
 ↪  */
.text
.global bad_mixed_read
.type   bad_mixed_read, %function
bad_mixed_read:
    /* WRONG: interpreting x1 as holding a double bits argument */
    /* This violates the ABI mapping and yields nonsense */
    add     x0, x0, x1
    ret
```

## 3.4 ABI Guarantees for Vector Register Preservation

### 3.4.1 What the ABI Actually Guarantees

At a call boundary, the ABI guarantees **preservation only for the registers designated as callee-saved**. Everything else is assumed volatile and may be clobbered by the call. Therefore:

- you may keep long-lived FP/vector locals in preserved SIMD registers *only if* you follow save/restore rules,
- you must assume argument/temporary SIMD registers can be overwritten by any call,
- and you must preserve alignment and proper stack space when spilling 128-bit vectors.

### 3.4.2 Example 9: Preserving a Vector Across a Call

```
/* Preserve a 128-bit vector value across a call using stack storage
 ↪  */
.text
.global preserve_vector_across_call
.type   preserve_vector_across_call, %function
preserve_vector_across_call:
    /* q0 holds a live vector value we need after calling helper_vec
     ↪  */
    sub     sp, sp, #32
    str     q0, [sp, #0]        /* save 16 bytes */
    /* Keep additional space for alignment and future growth */

    bl      helper_vec

    ldr     q0, [sp, #0]        /* restore live vector */
    add     sp, sp, #32
    ret
```

### 3.4.3 Example 10: ABI-Safe Rule Set for SIMD in Handwritten Assembly

- Treat `v0--v7` as **argument/return/volatile** registers: do not rely on them surviving calls.
- If you must keep a vector value across a call, **spill it** (stack) or move it into a preserved SIMD register and **save/restore** that preserved register.
- Always allocate stack space in multiples of **16 bytes** and keep `sp` aligned at call boundaries.

- When storing vectors, use `qN` view (`str/ldr qN`) and ensure proper alignment and space.

### 3.4.4 One-Page Practical Template (Non-Leaf FP Function)

```
/* Template: non-leaf FP function that needs to keep d0 across a call
 ↪  */
.text
.global fp_nonleaf_template
.type   fp_nonleaf_template, %function
fp_nonleaf_template:
    sub     sp, sp, #16
    str     d0, [sp, #0]        /* save incoming d0 */

    bl      helper_fp           /* may clobber v0.. */

    ldr     d0, [sp, #0]        /* restore d0 */
    add     sp, sp, #16
    fadd    d0, d0, d0
    ret
```

# Chapter 4

# Stack Layout and Alignment Rules

## 4.1 Stack Growth Direction and Alignment Requirements

### 4.1.1 Downward-Growing Stack (Architectural Convention)

In AArch64 user-space code following AAPCS64, the stack is a contiguous memory region addressed by `SP` that conventionally **grows toward lower addresses**:

- allocating stack space subtracts from `SP`,
- deallocating stack space adds to `SP`.

This convention enables efficient call nesting, local storage, register spills, and interoperation with debuggers and unwinders.

### 4.1.2 What the ABI Requires vs What the CPU Allows

The CPU allows many address computations, but the ABI imposes constraints so all components (compiler, assembler, linker, debugger, unwinder, sanitizer) agree on:

- when the stack is valid,
- how it is aligned,
- and what can be assumed at call boundaries.

### 4.1.3 Example 1: Minimal Stack Allocation Pattern

```
/* Allocate 16 bytes, use it, then deallocate (aligned and
 ↪  reversible) */
.text
.global stack_alloc_16
.type   stack_alloc_16, %function
stack_alloc_16:
    sub     sp, sp, #16
    /* local storage: [sp, #0..15] */
    add     x0, x0, #1
    add     sp, sp, #16
    ret
```

## 4.2 Mandatory 16-Byte Stack Alignment

### 4.2.1 The Non-Negotiable Rule

AAPCS64 requires that **`SP` be 16-byte aligned at public call boundaries**. Practically:

- On entry to a function (as called by conforming code), `SP % 16 == 0`.
- Before executing `bl` to call another function, `SP % 16 == 0` must still hold.
- Any stack allocation must keep alignment intact, so allocate in multiples of 16 bytes.

This is a correctness rule, not an optimization hint. Many implementations assume this alignment for efficient spills and for correct behavior of some instructions and runtime components.

### 4.2.2 Example 2: Correct Non-Leaf Function (Keeps Alignment)

```
/* Non-leaf: calls another function and preserves 16-byte alignment
 ↪  */
.text
.global nonleaf_aligned
.type   nonleaf_aligned, %function
nonleaf_aligned:
    sub     sp, sp, #16
    str     x30, [sp, #8]       /* save LR if needed after the call
     ↪  */

    bl      helper              /* SP is aligned here */

    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

### 4.2.3 Example 3: Misalignment Bug That Often Appears “Random”

```
/* BAD: subtracts 8 -> SP misaligned, then calls helper */
.text
.global nonleaf_misaligned_bug
.type   nonleaf_misaligned_bug, %function
nonleaf_misaligned_bug:
```

```
sub     sp, sp, #8          /* ABI violation: SP no longer
 ↪  16-byte aligned */
bl      helper              /* Undefined behavior at the ABI
 ↪  boundary */
add     sp, sp, #8
ret
```

### 4.2.4 Practical Rule

If you write assembly that calls anything:

- allocate stack space in multiples of **16 bytes**,
- and check alignment before `bl`.

## 4.3 Stack Frame Structure (Conceptual)

A stack frame is the region of stack memory a function uses to manage:

- preserved registers (callee-saved) it touches,
- saved return address (`LR`) if it is non-leaf or if needed for debugging/unwinding,
- local variables and temporary spill slots,
- and outgoing stack arguments for calls (when register arguments overflow).

### 4.3.1 Conceptual Layout (One Common Pattern)

A typical frame (conceptual) may contain:

- **saved FP/LR** (if a frame pointer is used),

- **callee-saved GPRs** (`x19--x28` as needed),
- **callee-saved SIMD** (`v8--v15` as needed, stored as 128-bit),
- **locals/spills** (compiler-chosen layout),
- optional **outgoing argument area** (if needed).

The ABI does not force a single exact layout for all frames; it forces the **observable correctness properties**:

- preserved registers must be restored,
- stack must be properly aligned at calls,
- and `SP` must be restored on return.

### 4.3.2 Example 4: Canonical Save/Restore of FP and LR

When a function establishes a conventional frame pointer, a common pattern is saving `x29` (FP) and `x30` (LR), then setting FP.

```
/* Conceptual framed function (structure example, not the only valid
 ↪  one) */
.text
.global framed_function
.type   framed_function, %function
framed_function:
    sub     sp, sp, #16
    stp     x29, x30, [sp, #0]  /* save FP and LR */
    mov     x29, sp             /* establish frame pointer */

    /* body */
```

```
    add     x0, x0, #7

    ldp     x29, x30, [sp, #0]  /* restore FP and LR */
    add     sp, sp, #16
    ret
```

### 4.3.3 Example 5: Saving Multiple Callee-Saved Registers

```
/* Save/restore callee-saved registers in aligned pairs */
.text
.global save_x19_x20
.type   save_x19_x20, %function
save_x19_x20:
    sub     sp, sp, #32
    stp     x19, x20, [sp, #0]  /* preserve regs if used */
    stp     x29, x30, [sp, #16] /* optional frame bookkeeping */
    mov     x29, sp

    /* body: use x19/x20 safely */
    mov     x19, x0
    mov     x20, x1
    add     x0, x19, x20

    ldp     x29, x30, [sp, #16]
    ldp     x19, x20, [sp, #0]
    add     sp, sp, #32
    ret
```

## 4.4 Red Zone: Why It Does Not Exist in AArch64

### 4.4.1 What "Red Zone" Means (Concept)

A "red zone" is an ABI feature where a fixed area below the current stack pointer may be used by leaf functions without adjusting `SP`. Some ABIs allow it to avoid stack pointer updates.

### 4.4.2 AAPCS64 Policy: Do Not Assume Untouched Space Below SP

In AAPCS64 practice, you must not assume that memory below `SP` is safe to use without allocating it, because:

- asynchronous events (interrupts, exceptions, signals) and runtime mechanisms may use the stack,
- system software may clobber below `SP` when handling an event,
- and portable ABI-correct code must treat `SP` as the boundary of valid stack allocation.

Therefore, **there is no red-zone rule you can rely on for correctness** in AArch64 AAPCS64 user-space interoperability. If you need stack storage, you must allocate it by adjusting `SP`.

### 4.4.3 Example 6: The Temptation (Do Not Do This)

```
/* BAD idea: using memory below SP without allocating it */
.text
.global redzone_like_bug
.type   redzone_like_bug, %function
redzone_like_bug:
    /* ABI-unsafe: writing below SP without reserving space */
    str     x0, [sp, #-8]       /* do not rely on this being safe */
```

```
    ldr     x0, [sp, #-8]
    ret
```

### 4.4.4 Example 7: Correct Replacement (Allocate and Use)

```
/* GOOD: allocate 16 bytes (aligned), then use it */
.text
.global safe_local_store
.type   safe_local_store, %function
safe_local_store:
    sub     sp, sp, #16
    str     x0, [sp, #0]
    ldr     x0, [sp, #0]
    add     sp, sp, #16
    ret
```

### 4.4.5 ABI Discipline Summary

- **No red-zone assumptions.**
- **If you store to the stack, reserve space first.**
- **Keep `SP` 16-byte aligned at call boundaries.**
- **Restore `SP` exactly before returning.**

# Chapter 5

# Function Prologue and Epilogue Mechanics

## 5.1 Minimal Leaf Function Frames

A **leaf function** does not call any other function. Under AAPCS64, the safest and fastest leaf is one that:

- uses only caller-saved registers (`x0--x18` as volatile locals),
- does not require stack locals or spills,
- and therefore needs **no stack frame at all**.

### 5.1.1 Example 1: Zero-Frame Leaf (Best Case)

```
/* Leaf: no calls, no stack, returns in x0 */
.text
.global leaf_add8
```

```
.type   leaf_add8, %function
leaf_add8:
    add     x0, x0, #8
    ret
```

### 5.1.2 When a Leaf Still Needs a Frame

A leaf may still require a frame if it:

- needs local stack storage (arrays, spills, large temporaries),
- uses callee-saved registers that must be preserved,
- or must keep strict unwind/debug conventions in a given build mode.

### 5.1.3 Example 2: Leaf With Stack Locals (Aligned)

```
/* Leaf: uses stack local storage, keeps SP aligned */
.text
.global leaf_with_locals
.type   leaf_with_locals, %function
leaf_with_locals:
    sub     sp, sp, #16
    str     x0, [sp, #0]        /* local slot */
    ldr     x1, [sp, #0]
    add     x0, x1, #1
    add     sp, sp, #16
    ret
```

## 5.2 Non-Leaf Function Stack Frames

A **non-leaf function** calls other functions. The moment you execute `bl`, you must assume:

- caller-saved registers may be clobbered by the callee,
- `LR` (`x30`) is overwritten by the *next* call,
- and `SP` must remain **16-byte aligned at the call boundary**.

### 5.2.1 Minimal Non-Leaf Requirement: Protect the Return Address

If a function calls another function and then needs to return, it must ensure its own return address is not lost. Common strategies:

- save `LR` on the stack,
- or move it into a callee-saved register (and preserve that register),
- or establish a standard frame with `FP/LR` save.

### 5.2.2 Example 3: Minimal Non-Leaf Saving Only LR (Aligned)

```
/* Non-leaf: saves LR because bl overwrites x30 */
.text
.global nonleaf_save_lr
.type   nonleaf_save_lr, %function
nonleaf_save_lr:
    sub     sp, sp, #16
    str     x30, [sp, #8]       /* save LR in aligned frame */

    bl      helper              /* may clobber volatile regs */
```

```
    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

### 5.2.3 Example 4: Non-Leaf With Locals and Saved Registers

```
/* Non-leaf: keeps a local across a call; uses stack slots */
.text
.global nonleaf_keep_local
.type   nonleaf_keep_local, %function
nonleaf_keep_local:
    sub     sp, sp, #32
    str     x30, [sp, #24]      /* save LR */
    str     x0,  [sp, #0]       /* save input local */

    bl      helper

    ldr     x1,  [sp, #0]       /* restore local */
    add     x0,  x1, #1
    ldr     x30, [sp, #24]
    add     sp, sp, #32
    ret
```

## 5.3 Saving and Restoring Registers Correctly

### 5.3.1 The Rule

**If you modify a callee-saved register, you must restore it before returning.**

This includes:

- preserved GPRs used as locals (commonly `x19--x28`),
- preserved SIMD registers when used as locals,
- and frame-related registers if you establish a conventional frame.

### 5.3.2 Correctness Over Style

The ABI does not require a single instruction pattern, but it requires the observable effect:

- on return, preserved registers must hold the same values they had at entry,
- `SP` must be restored exactly,
- and the function must branch to the correct return address.

### 5.3.3 Example 5: Correct Save/Restore of a Callee-Saved GPR

```
/* Uses x19 as a long-lived local; saves/restores it */
.text
.global use_x19_correctly
.type   use_x19_correctly, %function
use_x19_correctly:
    sub     sp, sp, #16
    str     x19, [sp, #0]       /* save preserved register */

    mov     x19, x0
    add     x19, x19, #42
    mov     x0, x19             /* return value */
```

```
    ldr     x19, [sp, #0]       /* restore preserved register */
    add     sp, sp, #16
    ret
```

### 5.3.4 Example 6: Saving/Restoring Multiple Registers Efficiently

Paired load/store reduces instruction count and naturally fits aligned frame layouts.

```
/* Save/restore two preserved registers as a pair */
.text
.global save_pair_x19_x20
.type   save_pair_x19_x20, %function
save_pair_x19_x20:
    sub     sp, sp, #16
    stp     x19, x20, [sp, #0]

    /* body: safe use */
    add     x19, x0, #1
    add     x20, x1, #2
    add     x0,  x19, x20

    ldp     x19, x20, [sp, #0]
    add     sp, sp, #16
    ret
```

### 5.3.5 Example 7: Common Bug — Save Without Restore (Silent Corruption)

```
/* BAD: saves x19 but never restores it */
.text
```

```
.global bad_missing_restore
.type   bad_missing_restore, %function
bad_missing_restore:
    sub     sp, sp, #16
    str     x19, [sp, #0]
    mov     x19, #7
    add     x0, x0, #1
    add     sp, sp, #16
    ret                         /* ABI violation: x19 modified */
```

### 5.3.6 A Robust Discipline Template

If your handwritten assembly is part of a larger system, adopt a template mindset:

- declare which preserved registers you will use,
- save them once in the prologue,
- restore them once in the epilogue,
- never early-return without restoring.

## 5.4 Frame Pointer (X29) Usage and Optionality

### 5.4.1 What X29 Means in Practice

`x29` is commonly used as the **frame pointer (FP)** when a function establishes a conventional frame. This provides:

- stable addressing for locals and saved registers,
- easier debugging and stack walking,

- more predictable unwind metadata generation in some build modes.

### 5.4.2 Optionality: You Can Omit FP

In optimized builds, compilers often omit the frame pointer and address locals relative to `SP` directly. This is valid as long as:

- the function preserves ABI-required registers,
- `SP` alignment is maintained at call boundaries,
- and `SP` is restored exactly on return.

### 5.4.3 Example 8: Conventional FP/LR Frame Setup

```
/* Frame-based function: saves FP/LR, sets FP */
.text
.global fp_based_function
.type   fp_based_function, %function
fp_based_function:
    sub     sp, sp, #16
    stp     x29, x30, [sp, #0]  /* save FP and LR */
    mov     x29, sp             /* set FP */

    /* body */
    add     x0, x0, #3

    ldp     x29, x30, [sp, #0]
    add     sp, sp, #16
    ret
```

### 5.4.4 Example 9: FP-Omitted Function (SP-Relative Only)

```
/* No FP: SP-relative addressing only */
.text
.global no_fp_function
.type   no_fp_function, %function
no_fp_function:
    sub     sp, sp, #16
    str     x30, [sp, #8]       /* save LR if needed */
    /* body */
    add     x0, x0, #3
    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

### 5.4.5 Choosing FP vs No FP (Engineering Guidance)

- Use **FP** when you prioritize debugging clarity, stable frame structure, and predictable stack walking.
- Omit **FP** when optimizing for register availability and when your build/debug strategy supports it.
- In handwritten assembly for libraries, a conservative default is to use a clear, consistent frame template unless performance demands otherwise.

# Chapter 6

# Argument Passing Rules in Detail

## 6.1 Integer and Pointer Arguments

### 6.1.1 Core Rule

For AAPCS64, the first integer/pointer arguments are passed in `x0--x7`. When there are more than available registers, remaining arguments are passed on the stack in ABI-defined slots. The callee must treat:

- `x0--x7` as the primary incoming integer/pointer argument locations,
- stack-passed arguments as valid only at the correct stack offsets (after stack setup),
- and must preserve stack alignment and any callee-saved registers it uses.

### 6.1.2 Example 1: Up to Eight Integer Arguments (All in Registers)

```
/* C/C++ */
#include <cstdint>
```

```
extern "C" std::uint64_t sum8(std::uint64_t a0, std::uint64_t a1,
↪  std::uint64_t a2, std::uint64_t a3,
                              std::uint64_t a4, std::uint64_t a5,
                               ↪  std::uint64_t a6, std::uint64_t a7);
```

```
/* AArch64 GAS: a0..a7 in x0..x7, return in x0 */
.text
.global sum8
.type   sum8, %function
sum8:
    add     x0, x0, x1
    add     x0, x0, x2
    add     x0, x0, x3
    add     x0, x0, x4
    add     x0, x0, x5
    add     x0, x0, x6
    add     x0, x0, x7
    ret
```

### 6.1.3 Example 2: Pointer Arguments Are Just Integer-Class

Pointers follow the same rule as integers: the first pointer arguments arrive in `x0--x7`.

```
/* C/C++ */
#include <cstdint>

extern "C" std::uint64_t load_add(const std::uint64_t* p, std::uint64_t
↪  addend);
```

```
/* p in x0, addend in x1 */
.text
```

```
.global load_add
.type   load_add, %function
load_add:
    ldr     x2, [x0]            /* load *p */
    add     x0, x2, x1          /* return = *p + addend */
    ret
```

### 6.1.4 Example 3: More Than Eight Integer Arguments (Register + Stack Concept)

When integer/pointer arguments exceed `x0--x7`, the overflow is passed on the stack. The correct stack offset depends on the caller's outgoing argument area and the callee frame. This booklet shows the concept and provides a safe inspection method.

```
/* C/C++ concept */
#include <cstdint>

extern "C" std::uint64_t sum10(std::uint64_t a0, std::uint64_t a1,
↪  std::uint64_t a2, std::uint64_t a3,
                               std::uint64_t a4, std::uint64_t a5,
                               ↪  std::uint64_t a6, std::uint64_t a7,
                               std::uint64_t a8, std::uint64_t a9);
```

```
/* Concept-only: a0..a7 in x0..x7; a8 and a9 are stack-passed.
   Exact offsets depend on the ABI stack argument area and current SP
   ↪  after any frame setup.
   In practice, prefer a C wrapper or inspect compiler-generated code
   ↪  for the specific platform. */
.text
.global sum10
.type   sum10, %function
```

```
sum10:
    /* sum registers first */
    add     x0, x0, x1
    add     x0, x0, x2
    add     x0, x0, x3
    add     x0, x0, x4
    add     x0, x0, x5
    add     x0, x0, x6
    add     x0, x0, x7
    /* stack loads for a8/a9 would follow here (platform-specific
     ↪  offset details) */
    ret
```

## 6.2 Floating-Point Arguments

### 6.2.1 Core Rule

Floating-point scalar arguments (`float`, `double`) are passed in `v0`--`v7` using `s0/s1/...` or `d0/d1/...`. Floating-point return values are returned in `v0` (`s0` or `d0`).

### 6.2.2 Example 4: FP Arguments and Return (double)

```
/* C/C++ */
extern "C" double hypot2(double x, double y);
```

```
/* x in d0, y in d1; return in d0 */
.text
.global hypot2
.type   hypot2, %function
hypot2:
```

```
    fmul    d2, d0, d0
    fmul    d3, d1, d1
    fadd    d0, d2, d3
    fsqrt   d0, d0
    ret
```

### 6.2.3 Example 5: More Than Eight FP Arguments (Register + Stack Concept)

FP arguments beyond `v0--v7` are passed via the stack according to ABI rules. As with integer overflow, exact offsets depend on the call frame layout.

## 6.3 Struct and Aggregate Passing Rules

### 6.3.1 Concept: Classification Determines Where It Goes

AAPCS64 classifies aggregates (structs/unions) to decide whether they are passed:

- in registers (possibly split across integer and/or FP registers),
- or by reference via a hidden pointer (indirect passing).

The ABI goal is efficiency without ambiguity: small aggregates may travel in registers; large aggregates travel indirectly.

### 6.3.2 Safe Engineering Guidance

Because aggregate classification is subtle and depends on exact layout, alignment, and member types, the safest interop strategy is:

- **do not guess** register mapping for structs in handwritten assembly,

- use a C wrapper that breaks aggregates into scalar arguments,
- or keep the assembly boundary scalar-only and reconstruct aggregates in C/C++.

### 6.3.3 Example 6: Scalar-Only Wrapper for a Struct Argument

```cpp
/* C/C++: struct passed at API level, but assembly boundary uses scalars */
#include <cstdint>

struct PairU64 { std::uint64_t a; std::uint64_t b; };

extern "C" std::uint64_t pair_sum_u64(std::uint64_t a, std::uint64_t b);

extern "C" std::uint64_t pair_sum(PairU64 p) {
    return pair_sum_u64(p.a, p.b);
}
```

```asm
/* Assembly boundary: scalar-only, stable mapping */
.text
.global pair_sum_u64
.type   pair_sum_u64, %function
pair_sum_u64:
    /* a in x0, b in x1 */
    add     x0, x0, x1
    ret
```

### 6.3.4 Example 7: Struct Return via Scalar Boundary

```cpp
/* Return two values via scalar returns or out-params instead of struct
↪  return */
#include <cstdint>
```

```
extern "C" void divmod_u64(std::uint64_t x, std::uint64_t y,
                           std::uint64_t* q_out, std::uint64_t* r_out);
```

```
/* x in x0, y in x1, q_out in x2, r_out in x3 */
.text
.global divmod_u64
.type   divmod_u64, %function
divmod_u64:
    udiv    x4, x0, x1          /* q = x / y */
    msub    x5, x4, x1, x0      /* r = x - q*y */
    str     x4, [x2]
    str     x5, [x3]
    ret
```

# 6.4 Large Objects and Indirect Passing

## 6.4.1 The Indirect Passing Model

When an object is too large or otherwise classified as non-register-passable, the ABI uses **indirect passing**:

- the caller allocates the object in memory,
- passes a pointer to it as an argument (in an integer register if available),
- and the callee accesses the object through that pointer.

This strategy is also used for some large returns: a hidden “sret” pointer is passed so the callee writes the return object into caller-provided memory.

### 6.4.2 Example 8: Explicit Indirect Passing Using Pointers (Recommended)

```cpp
/* C/C++: explicit pointer boundary for large data */
#include <cstdint>

struct Big {
    std::uint64_t v[8];
};

extern "C" std::uint64_t sum_big(const Big* p);
```

```asm
/* p in x0 */
.text
.global sum_big
.type   sum_big, %function
sum_big:
    ldr     x1, [x0, #0]
    ldr     x2, [x0, #8]
    ldr     x3, [x0, #16]
    ldr     x4, [x0, #24]
    ldr     x5, [x0, #32]
    ldr     x6, [x0, #40]
    ldr     x7, [x0, #48]
    ldr     x8, [x0, #56]
    add     x0, x1, x2
    add     x0, x0, x3
    add     x0, x0, x4
    add     x0, x0, x5
    add     x0, x0, x6
    add     x0, x0, x7
    add     x0, x0, x8
```

```
    ret
```

# 6.5 Variadic Functions (va_list Model)

## 6.5.1 Why Variadic Calls Are Special

Variadic functions (e.g., `printf`-style) cannot rely solely on the fixed signature, because additional unnamed arguments follow ABI-defined rules:

- some arguments arrive in registers,
- some may be on the stack,
- and the callee must use a `va_list` mechanism to retrieve them correctly.

## 6.5.2 The Practical Rule for Interop

Do not implement variadic functions in handwritten assembly unless you fully implement the platform's `va_list` layout and the ABI-required register save areas. The robust interop approach is:

- keep variadic logic in C/C++,
- expose a non-variadic, explicitly-typed wrapper for assembly boundaries.

## 6.5.3 Example 9: Safe Wrapper Pattern Around Variadic

```
/* C/C++: variadic stays in C; assembly boundary is fixed */
#include <cstdint>
#include <cstdarg>

static std::uint64_t sum_va(int n, ...) {
```

```
    std::va_list ap;
    va_start(ap, n);
    std::uint64_t s = 0;
    for (int i = 0; i < n; ++i) {
        s += va_arg(ap, std::uint64_t);
    }
    va_end(ap);
    return s;
}

extern "C" std::uint64_t sum3_u64(std::uint64_t a, std::uint64_t b,
↪   std::uint64_t c) {
    return sum_va(3, a, b, c);
}
```

```
/* Assembly-friendly fixed-arity boundary: a,b,c in x0,x1,x2 */
.text
.global sum3_u64_asm
.type   sum3_u64_asm, %function
sum3_u64_asm:
    add     x0, x0, x1
    add     x0, x0, x2
    ret
```

### 6.5.4 Key Takeaways for Variadics

- Variadic retrieval depends on ABI-defined `va_list` structure and register spill areas.
- Cross-module correctness requires using the compiler's definition of `va_list`.
- For assembly interoperability, prefer fixed-arity wrappers and scalar-only boundaries.

# Chapter 7

# Return Value Rules

## 7.1 Scalar Return Values

### 7.1.1 Core Rule

In AAPCS64, the primary return location for **integer and pointer scalars** is `x0`. For common scalar sizes (up to 64-bit), the callee places the result in `x0` and returns with `ret`. For multi-register scalar returns (rare in stable interop boundaries), additional registers may be used by ABI rule, but robust interop boundaries prefer explicit, simple return forms.

### 7.1.2 Example 1: Return `uint64_t` in `x0`

```
/* C/C++ */
#include <cstdint>
extern "C" std::uint64_t add_u64(std::uint64_t a, std::uint64_t b);

std::uint64_t demo(std::uint64_t x, std::uint64_t y) {
    return add_u64(x, y);
```

```
}
```

```
/* a in x0, b in x1, return in x0 */
.text
.global add_u64
.type   add_u64, %function
add_u64:
    add     x0, x0, x1
    ret
```

### 7.1.3 Example 2: Return a Pointer in `x0`

```
/* C/C++ */
#include <cstdint>
extern "C" const std::uint64_t* next_ptr(const std::uint64_t* p);
```

```
/* p in x0, return pointer in x0 */
.text
.global next_ptr
.type   next_ptr, %function
next_ptr:
    add     x0, x0, #8
    ret
```

### 7.1.4 Example 3: 32-bit Return Values

A 32-bit integer return uses `w0` (the lower 32 bits of `x0`) as the return location.

```
/* C/C++ */
#include <cstdint>
extern "C" std::uint32_t add_u32(std::uint32_t a, std::uint32_t b);
```

```
/* a in w0, b in w1, return in w0 */
.text
.global add_u32
.type   add_u32, %function
add_u32:
    add     w0, w0, w1
    ret
```

## 7.2 Floating-Point Return Values

### 7.2.1 Core Rule

Floating-point scalar returns use the FP/SIMD register bank:

- `float` returns in `s0`,
- `double` returns in `d0`,
- and the architectural container is `v0`.

### 7.2.2 Example 4: Return `double` in `d0`

```
/* C/C++ */
extern "C" double mul_add(double x, double y, double z);
```

```
/* x in d0, y in d1, z in d2, return in d0 */
.text
.global mul_add
.type   mul_add, %function
mul_add:
    fmul    d0, d0, d1
```

```
    fadd    d0, d0, d2
    ret
```

### 7.2.3 Example 5: Return `float` in `s0`

```
/* C/C++ */
extern "C" float half(float x);
```

```
/* x in s0, return in s0 */
.text
.global half
.type   half, %function
half:
    fmov    s1, #0.5
    fmul    s0, s0, s1
    ret
```

## 7.3 Struct and Aggregate Returns

### 7.3.1 Concept: Small Aggregates May Return in Registers

AAPCS64 may return some small aggregates in registers (potentially using `x0--x1` and/or `v0--v1` depending on classification). However, aggregate classification is subtle and depends on:

- size,
- alignment,
- member types (integer vs FP),
- and platform/toolchain ABI details at the language boundary.

### 7.3.2 Engineering Rule for Interoperability

For **handwritten assembly interoperability**, do not rely on implicit aggregate-return classification unless you lock down the exact ABI and validate against compiler output for your target. Instead, prefer one of:

- return scalars (or multiple scalars) explicitly,
- or use an explicit out-parameter (pointer) for structured outputs.

### 7.3.3 Example 6: Replace Struct Return With Out-Parameters (Recommended)

```
/* C/C++: stable boundary */
#include <cstdint>

extern "C" void split_u64(std::uint64_t x, std::uint32_t* lo, std::uint32_t*
↪  hi);
```

```
/* x in x0, lo* in x1, hi* in x2 */
.text
.global split_u64
.type   split_u64, %function
split_u64:
    /* lo = low 32 bits */
    str     w0, [x1]
    /* hi = high 32 bits */
    lsr     x3, x0, #32
    str     w3, [x2]
    ret
```

### 7.3.4 Example 7: Two-Value Return via Two Scalars

When you control both sides, you can return a second scalar via an out-pointer or by returning one value and writing the other. This pattern is ABI-stable and easy to debug.

```
/* C/C++ */
#include <cstdint>
extern "C" std::uint64_t divrem_u64(std::uint64_t x, std::uint64_t y,
↪  std::uint64_t* rem_out);
```

```
/* x in x0, y in x1, rem_out in x2; return quotient in x0 */
.text
.global divrem_u64
.type   divrem_u64, %function
divrem_u64:
    udiv    x3, x0, x1          /* q */
    msub    x4, x3, x1, x0      /* r = x - q*y */
    str     x4, [x2]            /* *rem_out = r */
    mov     x0, x3              /* return q */
    ret
```

## 7.4 Hidden Return Pointers

### 7.4.1 The “sret” Model (Indirect Return)

When an aggregate return is not suitable for register return, the ABI uses an **indirect return**:

- the caller allocates storage for the return object,
- passes a hidden pointer to that storage to the callee,
- the callee writes the result into that memory and returns normally.

This is often called an “sret” (structure return) mechanism. The **key takeaway** is that the return object may not be returned in registers at all; it is **materialized in caller-provided memory**.

### 7.4.2 Interop Guidance

For handwritten assembly boundaries:

- treat indirect returns as a pointer-based protocol,
- prefer making it explicit in the API rather than relying on a hidden pointer,
- because explicit out-pointers are clearer, easier to audit, and stable across toolchains.

### 7.4.3 Example 8: Explicit “Return Object” Pointer (Recommended Replacement)

```cpp
/* C/C++ explicit replacement for hidden sret */
#include <cstdint>

struct Pair {
    std::uint64_t a;
    std::uint64_t b;
};

extern "C" void make_pair(std::uint64_t x, std::uint64_t y, Pair* out);
```

```asm
/* x in x0, y in x1, out in x2 */
.text
.global make_pair
.type   make_pair, %function
make_pair:
```

```
    str     x0, [x2, #0]
    str     x1, [x2, #8]
    ret
```

### 7.4.4 Example 9: Returning a Vector Result Safely

Even if a platform/toolchain may support returning small vectors in registers, stable interop often prefers:

- return status in `x0`,
- write vector results to caller-provided memory.

```
/* C/C++ */
#include <cstdint>

extern "C" int compute_vec2(double x, double y, double* out2); /*
↪  out2[0..1] */
```

```
/* x in d0, y in d1, out2 in x0 (first integer reg because signature
↪  places it there) */
.text
.global compute_vec2
.type   compute_vec2, %function
compute_vec2:
    /* Store results explicitly */
    fadd    d2, d0, d1
    fsub    d3, d0, d1
    str     d2, [x0, #0]
    str     d3, [x0, #8]
    mov     w0, #0              /* return status = 0 */
    ret
```

### 7.4.5 Summary Discipline

- Scalar integer/pointer returns: `x0` (`w0` for 32-bit).
- Scalar FP returns: `s0`/`d0` (in `v0`).
- Aggregate returns: may be registers for small cases, but interop-safe design prefers explicit scalar/out-pointer boundaries.
- Hidden return pointers exist for large aggregates; making them explicit improves correctness and auditability.

# Chapter 8

# C and C++ Interoperability Rules

## 8.1 Name Mangling vs ABI Compatibility

### 8.1.1 Two Different Problems: Calling Convention vs Linkage Encoding

AAPCS64 defines the **calling convention** (register/stack rules). C++ adds another layer: **name mangling**, which is how the compiler encodes:

- namespaces,
- classes,
- overload sets,
- templates,
- and parameter types

into symbol names.
Therefore, two functions can be **AAPCS64-compatible at the call boundary** but **not link-compatible** if their symbols do not match.

### 8.1.2 What Breaks Most Often

- C++ overloads and templates generate different symbol names across compilers/toolchains.
- Minor type changes (e.g., `int` vs `long`) change mangled names and may also change ABI classification.
- Different compilation modes or ABI variants can produce incompatible C++ ABIs even on the same ISA.

### 8.1.3 Example 1: Overloads Cannot Share One Stable Symbol Name

```
/* C++ */
int f(int);
double f(double);

/* These are two different symbols in the object file (mangled names). */
```

An assembly file cannot reliably call `f` by writing `bl f` because there is no single unmangled `f` symbol.

## 8.2 `extern "C"` and Symbol Stability

### 8.2.1 What `extern "C"` Actually Does

`extern "C"` requests **C language linkage** for a declaration:

- disables C++ name mangling for the symbol name,
- enforces C linkage rules at the symbol level,

- and makes the function **callable by assembly and C** using a stable symbol identifier.

Important: `extern "C"` does not “switch off” C++ semantics inside the function body. It only stabilizes the **link boundary** and the signature form.

### 8.2.2 Example 2: Stable Entry Point for Assembly

```
/* C/C++ */
#include <cstdint>

extern "C" std::uint64_t add3_u64(std::uint64_t a, std::uint64_t b,
↪  std::uint64_t c);
```

```
/* Assembly can reliably call the symbol by name */
.text
.global call_add3
.type   call_add3, %function
call_add3:
    /* x0,x1,x2 already set by caller */
    bl      add3_u64
    ret
```

### 8.2.3 Example 3: Exposing a C ABI Wrapper Around C++ Implementation

```
/* C ABI boundary */
#include <cstdint>

static std::uint64_t impl(std::uint64_t x) {
    return x * 3 + 1;
}
```

```
extern "C" std::uint64_t stable_api(std::uint64_t x) {
    return impl(x);
}
```

This pattern is the default recommendation for long-lived binary interfaces.

## 8.3 Passing C++ Objects Across ABI Boundaries

### 8.3.1 The Interop Rule

Do not pass C++ objects by value across a binary boundary unless you control:

- the exact compiler and version,
- the same standard library implementation,
- the same build flags that affect ABI,
- and the same platform C++ ABI rules.

Reason: C++ object layout and calling conventions for class types depend on:

- padding and alignment,
- vtables and RTTI,
- small-string optimization details,
- exception/runtime models,
- and template instantiations.

### 8.3.2 Preferred Boundary: Opaque Handles + Explicit Functions

Use an opaque pointer (`void*`) as the ABI-stable handle and provide explicit create/destroy/operate functions.

### 8.3.3 Example 4: Opaque Handle Pattern (Recommended)

```cpp
/* Public C ABI */
#include <cstdint>
#include <new>

struct Obj {
    std::uint64_t bias;
    std::uint64_t compute(std::uint64_t x) const { return x + bias; }
};

extern "C" void* obj_create(std::uint64_t bias) {
    return new (std::nothrow) Obj{bias};
}

extern "C" void obj_destroy(void* p) {
    delete static_cast<Obj*>(p);
}

extern "C" std::uint64_t obj_compute(void* p, std::uint64_t x) {
    return static_cast<Obj*>(p)->compute(x);
}
```

Assembly (or other languages) can interoperate using only pointers and scalars.

### 8.3.4 Example 5: POD-Only Boundary (When You Must Pass a Struct)

If you must pass structured data, restrict it to **C-compatible POD** with fixed layout and no constructors, no virtual members, no references, and no hidden invariants.

```
/* POD boundary */
#include <cstdint>

struct PairU64 {
    std::uint64_t a;
    std::uint64_t b;
};

extern "C" std::uint64_t pair_sum(std::uint64_t a, std::uint64_t b);
```

```
/* Scalar boundary remains stable and unambiguous */
.text
.global pair_sum
.type   pair_sum, %function
pair_sum:
    add     x0, x0, x1
    ret
```

## 8.4 Constructors, Destructors, and ABI Constraints

### 8.4.1 Why Constructors/Destructors Are ABI-Hard

Constructors and destructors are not just functions:

- they may have hidden parameters (`this`, allocation context),
- they may participate in exception unwinding,

- they may require runtime support for virtual dispatch and base construction,
- and the emitted symbols and calling patterns are toolchain/platform C++ ABI details.

Therefore, calling C++ constructors/destructors directly from handwritten assembly as a stable interface is not recommended unless you fully lock the platform ABI and inspect compiler output.

### 8.4.2 Recommended Strategy

- Keep construction/destruction inside C++.
- Expose explicit `create`/`destroy` functions with `extern "C"` linkage.
- Ensure `destroy` is always called in the same binary domain that performed allocation if allocators may differ.

### 8.4.3 Example 6: Construction/Destruction Wrapper With Explicit Ownership

```
/* C ABI with explicit ownership */
#include <new>
#include <cstdint>

struct Buffer {
    std::uint64_t size;
    std::uint64_t* data;
    Buffer(std::uint64_t n) : size(n), data(new (std::nothrow)
    ↪  std::uint64_t[n]{}) {}
    ~Buffer() { delete[] data; }
};
```

```
extern "C" void* buffer_create(std::uint64_t n) {
    return new (std::nothrow) Buffer(n);
}

extern "C" void buffer_destroy(void* p) {
    delete static_cast<Buffer*>(p);
}
```

# 8.5 Exception Handling Boundaries (Conceptual)

## 8.5.1 Why Exceptions Are Not a Stable Binary Boundary

C++ exceptions require:

- language-specific runtime (unwind library),
- ABI-defined unwind tables and personality routines,
- consistent typeinfo/RTTI encoding across modules,
- and a stable contract for stack unwinding across call frames.

Even when the underlying unwind mechanism is standardized at a platform level, the **C++ language-level exception ABI** is toolchain- and platform-dependent.

## 8.5.2 Interop Rule: Do Not Let Exceptions Cross the Boundary

At a stable C ABI boundary:

- do not throw exceptions across it,
- catch exceptions inside the C++ side,

- translate them into error codes or explicit status objects,
- and keep the assembly/caller side exception-free.

### 8.5.3 Example 7: Translate Exceptions to Error Codes

```
/* Stable boundary: no exceptions escape */
#include <cstdint>

static std::uint64_t may_fail(std::uint64_t x) {
    if (x == 0) { throw 1; }
    return 100 / x;
}

extern "C" int safe_api(std::uint64_t x, std::uint64_t* out) {
    try {
        *out = may_fail(x);
        return 0;
    } catch (...) {
        *out = 0;
        return -1;
    }
}
```

### 8.5.4 Example 8: Assembly Caller Uses Status + Out-Value

```
/* Calls safe_api(x, &out). Returns status in w0; out written to
 ↪  memory. */
.text
.global asm_call_safe_api
.type   asm_call_safe_api, %function
asm_call_safe_api:
```

```
    /* x0 = input x, x1 = pointer to out */
    bl      safe_api
    /* w0 = status (0 success, -1 failure) */
    ret
```

### 8.5.5 Summary Discipline for Interop

- Use `extern "C"` for stable symbols callable from assembly/C.
- Prefer opaque handles and scalar-only APIs across binary boundaries.
- Keep constructors/destructors inside C++ and wrap them with explicit C ABI functions.
- Do not allow exceptions to cross the boundary; translate to status codes.

# Chapter 9

# Compiler-Generated Code and ABI Guarantees

## 9.1 What Compilers Must Guarantee

A compiler targeting AAPCS64 must emit code that satisfies the ABI contract at every externally-visible call boundary. This is what enables separate compilation, linking, and interoperability with handwritten assembly.

### 9.1.1 Mandatory ABI Guarantees at Call Boundaries

For a conforming AAPCS64 interface, compilers must ensure:

- **Correct argument placement:** integer/pointer arguments are placed in `x0`–`x7` (and overflow on stack), FP arguments in `v0`–`v7` (and overflow on stack), following ABI classification rules.
- **Correct return placement:** integer/pointer results are returned in `x0`; FP results in `v0`

(`s0`/`d0`); aggregates follow ABI return rules or indirect return conventions.

- **Register preservation:** callee-saved registers retain their incoming values on return from a call.
- **Stack pointer alignment:** `SP` is 16-byte aligned at public interfaces and remains aligned at any call site (`bl`).
- **Stack restoration:** on function return, `SP` is restored to its entry value (modulo ABI-defined red-zone absence; i.e., do not rely on below-`SP` space).

### 9.1.2 Observable Guarantee vs Implementation Freedom

The ABI constrains **observable behavior**, not the exact instruction sequence. A compiler may choose different prologues, epilogues, and instruction schedules as long as the contract holds.

### 9.1.3 Example 1: ABI-Visible Function (Compiler Must Preserve Contract)

```
/* External interface: ABI-visible */
#include <cstdint>
extern "C" std::uint64_t api_add(std::uint64_t a, std::uint64_t b) {
    return a + b;
}
```

A compiler may:

- keep everything in registers,
- emit no stack frame at all,
- still satisfy: inputs in `x0`/`x1`, output in `x0`.

### 9.1.4 Example 2: When the Compiler Must Save/Restore

If an ABI-visible function uses callee-saved registers or calls another function, the compiler must preserve required state.

```
/* Non-leaf: forces LR management and may force spills */
#include <cstdint>
extern "C" std::uint64_t helper(std::uint64_t);

extern "C" std::uint64_t api_call(std::uint64_t x) {
    return helper(x) + 1;
}
```

The compiler must ensure:

- `SP` alignment at the call to `helper`,
- proper return behavior,
- and preservation of callee-saved registers it chooses to use.

## 9.2 What Programmers Must Never Assume

ABI failures in mixed C/C++/assembly often happen because programmers assume properties that compilers are **not required** to maintain.

### 9.2.1 Never Assume Specific Prologue/Epilogue Instructions

Do not assume:

- that a function always saves `x29/x30` in a specific way,
- that a frame pointer is always used,
- or that the stack frame has a fixed layout across builds.

### 9.2.2 Never Assume Volatile Registers Survive Calls

Do not assume any caller-saved register survives a call. If you need a value after calling something, preserve it.

```
/* BAD: assumes x0 remains unchanged after call */
.text
.global bad_assumption_x0
.type   bad_assumption_x0, %function
bad_assumption_x0:
    /* x0 = important value */
    bl      helper              /* may clobber x0 */
    add     x0, x0, #1           /* BUG: x0 may not be original */
    ret
```

### 9.2.3 Never Assume Stack Space Below SP Is Safe

Do not use memory below `SP` without allocating it. There is no red-zone you can rely on for portable correctness.

```
/* BAD: writes below SP without reserving */
.text
.global bad_below_sp
.type   bad_below_sp, %function
bad_below_sp:
    str     x0, [sp, #-8]
    ldr     x0, [sp, #-8]
    ret
```

### 9.2.4 Never Assume Struct Layout/Passing Without ABI Rules

Do not guess aggregate passing/returning rules in assembly. If you need stable interop:

- use scalar-only boundaries,
- or explicit pointers/out-parameters.

## 9.3 Optimization vs ABI Preservation

### 9.3.1 Optimization Changes Everything Except the Contract

Under optimization, compilers may:

- inline functions,
- remove stack frames,
- reorder computations,
- allocate locals to registers,
- eliminate loads/stores,
- and transform control flow.

But they must still preserve ABI invariants at externally-visible boundaries.

### 9.3.2 Example 3: Same Source, Two Different Valid Shapes

```
/* The compiler may emit very different machine code for different options
 ↪  */
#include <cstdint>
```

```
extern "C" std::uint64_t f(std::uint64_t x) {
    std::uint64_t t = x + 7;
    return t * 3;
}
```

Valid outcomes include:

- a leaf with no stack usage,
- a version using SIMD for strength reduction (if profitable),
- or a version using callee-saved registers if register pressure requires.

All are valid if: **input arrives as defined, output returns as defined, preserved registers remain preserved, and SP alignment rules are satisfied at calls.**

### 9.3.3 Example 4: ABI Boundary Wrapper Stabilizes Interop

When mixing with handwritten assembly, isolate optimization changes behind a stable wrapper boundary:

```
/* Wrapper: stable boundary for assembly and other modules */
#include <cstdint>

static std::uint64_t impl(std::uint64_t x) {
    return (x + 7) * 3; /* may be optimized freely */
}

extern "C" std::uint64_t stable_entry(std::uint64_t x) {
    return impl(x); /* ABI-visible function */
}
```

## 9.4 Inline Functions and ABI Transparency

### 9.4.1 Inlining Removes the Call Boundary

When a function is inlined, the **calling convention is no longer exercised** at that call site because there is no call. Consequences:

- register usage becomes a local optimization detail,
- stack frames may disappear,
- and assumptions based on "how the function is called" become invalid.

Therefore, **do not debug ABI by inspecting an inlined call path**. Ensure you inspect true call boundaries.

### 9.4.2 Example 5: Inline Changes Observability

```
/* Candidate for inlining */
static inline std::uint64_t add1(std::uint64_t x) { return x + 1; }

extern "C" std::uint64_t api(std::uint64_t a) {
    return add1(a); /* may inline -> no call boundary here */
}
```

In optimized builds, `add1` may vanish entirely. The ABI still matters at `api`'s boundary (entry/exit), not inside.

### 9.4.3 Example 6: Force a Real Boundary for ABI Testing

Use a separate translation unit, avoid `static inline`, or ensure the function has external linkage so the compiler cannot trivially inline it across units (still not guaranteed, but more testable).

```cpp
/* TU1 */
#include <cstdint>
extern "C" std::uint64_t boundary(std::uint64_t x) { return x + 1; }
```

```cpp
/* TU2 */
#include <cstdint>
extern "C" std::uint64_t boundary(std::uint64_t);
extern "C" std::uint64_t api2(std::uint64_t a) { return boundary(a); }
```

### 9.4.4 Example 7: Assembly Interop Requires Non-Inlined Stable Symbols

If assembly calls into C/C++, you need a stable symbol. Combine:

- `extern "C"` for stable linkage,
- and a non-header-defined function body for a robust boundary.

```asm
/* Assembly calling a stable external symbol */
.text
.global asm_calls_boundary
.type   asm_calls_boundary, %function
asm_calls_boundary:
    /* x0 holds input */
    bl      boundary
    ret
```

## 9.5 ABI Guarantee Checklist (Practical)

- Do not rely on specific prologue/epilogue patterns.
- Preserve your own live values across calls; assume volatile registers are clobbered.
- Keep `SP` 16-byte aligned at every `bl`.

- Do not assume below-`SP` space is safe.
- Avoid struct-by-value interop in handwritten assembly; prefer scalar/pointer boundaries.
- Remember: inlining removes call boundaries; debug ABI at true external boundaries.

# Chapter 10

# Interfacing Handwritten Assembly with C/C++

## 10.1 Writing ABI-Compliant Assembly Functions

Handwritten assembly is "ABI-correct" only if it behaves exactly like a compiler-generated function at the call boundary. This means your assembly must implement the same external contract:

- accept arguments in the ABI-defined locations,
- return results in the ABI-defined locations,
- preserve callee-saved registers you modify,
- keep `SP` 16-byte aligned at every call boundary,
- restore `SP` exactly on return,
- and return to the correct address.

### 10.1.1 Minimum ABI Checklist for an Assembly Function

- Read integer/pointer arguments from `x0--x7`.
- Read FP arguments from `v0--v7` (`s0/d0..`).
- Return integer/pointer in `x0`; return FP in `v0`.
- If you touch a callee-saved register (e.g., `x19--x28` or preserved SIMD regs), save/restore it.
- If you call anything (`bl`), save `LR` if you need it afterward.
- Keep `SP` aligned (allocate in multiples of 16 bytes).

### 10.1.2 Example 1: Leaf Function, No Frame (Best Interop Case)

```
/* C/C++ */
#include <cstdint>
extern "C" std::uint64_t u64_add8(std::uint64_t x);
```

```
/* ABI-compliant leaf: x0 in, x0 out */
.text
.global u64_add8
.type   u64_add8, %function
u64_add8:
    add     x0, x0, #8
    ret
```

### 10.1.3 Example 2: Leaf Function With Callee-Saved Local

```
/* C/C++ */
#include <cstdint>
extern "C" std::uint64_t add_bias(std::uint64_t x);
```

```
/* Uses x19 (callee-saved) correctly */
.text
.global add_bias
.type   add_bias, %function
add_bias:
    sub     sp, sp, #16
    str     x19, [sp, #0]       /* save preserved register */

    mov     x19, x0
    add     x19, x19, #123
    mov     x0, x19

    ldr     x19, [sp, #0]
    add     sp, sp, #16
    ret
```

### 10.1.4 Example 3: FP Function (double) ABI Boundary

```
/* C/C++ */
extern "C" double add_d(double a, double b);
```

```
/* a in d0, b in d1, return in d0 */
.text
.global add_d
.type   add_d, %function
add_d:
    fadd    d0, d0, d1
    ret
```

## 10.2 Calling C/C++ Functions from Assembly

### 10.2.1 The Contract When You Are the Caller

When assembly calls into C/C++ you must behave like a compiler-generated caller:

- place arguments in the correct registers (and stack for overflow),
- ensure `SP` is 16-byte aligned at the call instruction,
- assume caller-saved registers may be clobbered by the callee,
- preserve your own live values across the call (stack or callee-saved regs),
- handle return values from `x0` or `v0`.

### 10.2.2 Example 4: Assembly Calls a C Function (Scalar Only)

```
/* C/C++ */
#include <cstdint>
extern "C" std::uint64_t c_mul3(std::uint64_t x) { return x * 3; }
```

```
/* Assembly: calls c_mul3(x0) and then adds 1 */
.text
.global asm_calls_c_mul3
.type   asm_calls_c_mul3, %function
asm_calls_c_mul3:
    sub     sp, sp, #16
    str     x30, [sp, #8]       /* save LR for non-leaf */

    bl      c_mul3              /* x0 is argument and return */
```

```
    ldr     x30, [sp, #8]
    add     sp, sp, #16
    add     x0, x0, #1
    ret
```

### 10.2.3 Example 5: Preserving a Live Value Across the Call

```
/* Preserve x0 across a call (caller-saved) */
.text
.global asm_preserve_x0_then_call
.type   asm_preserve_x0_then_call, %function
asm_preserve_x0_then_call:
    sub     sp, sp, #32
    str     x30, [sp, #24]      /* save LR */
    str     x0,  [sp, #0]       /* save live x0 */

    bl      helper

    ldr     x0,  [sp, #0]       /* restore x0 */
    ldr     x30, [sp, #24]
    add     sp, sp, #32
    ret
```

### 10.2.4 Example 6: Mixed Integer and FP Call

```
/* C/C++ */
#include <cstdint>
extern "C" double mix_call(std::uint64_t a, double x, std::uint64_t b,
↪  double y);
```

```
/* Assembly caller: a->x0, x->d0, b->x1, y->d1; return d0 */
```

```
.text
.global asm_calls_mix_call
.type   asm_calls_mix_call, %function
asm_calls_mix_call:
    sub     sp, sp, #16
    str     x30, [sp, #8]

    /* Assume x0/x1 and d0/d1 are already set by our caller; forward
     ↪  them */
    bl      mix_call

    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

## 10.3 Common Prologue/Epilogue Templates

### 10.3.1 Template A: Leaf, No Frame (Fastest)

Use when:

- no calls,
- no stack locals,
- no callee-saved registers used.

```
.text
.global leaf_template
.type   leaf_template, %function
leaf_template:
```

```
    /* body: use only volatile regs */
    ret
```

### 10.3.2 Template B: Non-Leaf Minimal (Save LR Only)

Use when:

- you call another function,
- you do not need callee-saved registers or a frame pointer.

```
.text
.global nonleaf_lr_template
.type   nonleaf_lr_template, %function
nonleaf_lr_template:
    sub     sp, sp, #16
    str     x30, [sp, #8]

    /* body: may call other functions */
    /* bl some_function */

    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

### 10.3.3 Template C: Conventional Frame (Save FP/LR)

Use when:

- you want predictable stack walking / debugging,
- or you prefer consistent structure for complex functions.

```
.text
.global fp_lr_frame_template
.type   fp_lr_frame_template, %function
fp_lr_frame_template:
    sub     sp, sp, #16
    stp     x29, x30, [sp, #0]
    mov     x29, sp

    /* body */

    ldp     x29, x30, [sp, #0]
    add     sp, sp, #16
    ret
```

### 10.3.4 Template D: Uses Callee-Saved Locals (Plus Optional FP/LR)

Use when:

- you need a preserved register for long-lived locals,
- you must still preserve stack alignment.

```
.text
.global callee_saved_template
.type   callee_saved_template, %function
callee_saved_template:
    sub     sp, sp, #32
    stp     x19, x20, [sp, #0]
    stp     x29, x30, [sp, #16]
    mov     x29, sp
```

```
    /* body: safe use of x19/x20 */

    ldp     x29, x30, [sp, #16]
    ldp     x19, x20, [sp, #0]
    add     sp, sp, #32
    ret
```

## 10.4 Debugging ABI Violations

ABI violations are dangerous because they often:

- work at `-O0` and fail at `-O2/-O3`,
- disappear under debugging (different register allocation),
- appear as random crashes or data corruption far from the call site.

### 10.4.1 The Four Most Common Failure Classes

1. **Stack misalignment at a call** (`SP` not 16-byte aligned at `bl`).
2. **Callee-saved register clobber** (e.g., `x19` modified without restore).
3. **Wrong argument bank** (reading FP args from `xN` instead of `dN`, or vice versa).
4. **Wrong return location** (placing return in the wrong register).

### 10.4.2 Example 7: Stack Misalignment Bug (Diagnostic Pattern)

```
/* BAD: misaligns SP then calls */
.text
.global bug_misalign_then_call
```

```
.type   bug_misalign_then_call, %function
bug_misalign_then_call:
    sub     sp, sp, #8          /* ABI violation */
    bl      helper
    add     sp, sp, #8
    ret
```

### 10.4.3 Correct Fix

```
/* GOOD: allocate 16 bytes (aligned) */
.text
.global fix_align_then_call
.type   fix_align_then_call, %function
fix_align_then_call:
    sub     sp, sp, #16
    str     x30, [sp, #8]
    bl      helper
    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

### 10.4.4 Example 8: Callee-Saved Clobber Bug

```
/* BAD: clobbers x19 (callee-saved) */
.text
.global bug_clobber_x19
.type   bug_clobber_x19, %function
bug_clobber_x19:
    mov     x19, x0             /* ABI violation unless
     ↪  saved/restored */
```

```
    add     x0, x0, #1
    ret
```

### 10.4.5 Correct Fix

```
/* GOOD: save/restore x19 */
.text
.global fix_preserve_x19
.type   fix_preserve_x19, %function
fix_preserve_x19:
    sub     sp, sp, #16
    str     x19, [sp, #0]
    mov     x19, x0
    add     x0, x19, #1
    ldr     x19, [sp, #0]
    add     sp, sp, #16
    ret
```

### 10.4.6 A Practical Debugging Workflow (Tool-Agnostic)

- **Step 1: Confirm the signature.** Ensure your assembly matches the exact C/C++ prototype (types and order).
- **Step 2: Verify call boundary alignment.** Ensure `SP` is 16-byte aligned at every `bl`.
- **Step 3: Audit preserved registers.** List every callee-saved register your function touches; verify save/restore.
- **Step 4: Audit argument banks.** Scalars in `xN`, FP in `dN/sN`, vectors in `qN`.
- **Step 5: Inspect compiler output for a reference implementation.** Write the same function in C and compare its calling pattern and frame discipline.

- **Step 6: Re-test under optimization.** ABI bugs often appear only at `-O2`/`-O3`.

# Chapter 11

# Common Mistakes and Silent ABI Breakage

ABI bugs are dangerous because they often:

- produce **wrong results without crashing**,
- appear only under optimization,
- vanish when you add logging or debug prints (register allocation changes),
- and surface far away from the original mistake.

## 11.1 Stack Misalignment Failures

### 11.1.1 The Failure

AAPCS64 requires `SP` to be **16-byte aligned** at every public call boundary and at every call site (`bl`). Misalignment commonly happens when handwritten assembly allocates 8 bytes or

any non-multiple of 16 and then calls a function.

### 11.1.2 Symptoms

- crashes inside unrelated callees,
- wrong floating-point results,
- sporadic failures only under `-O2`/`-O3`,
- failures that disappear when you remove a call or add a local variable.

### 11.1.3 Bad Example (Misalign Then Call)

```
/* BAD: SP misaligned at call site */
.text
.global bug_misaligned_call
.type   bug_misaligned_call, %function
bug_misaligned_call:
    sub     sp, sp, #8          /* ABI violation */
    bl      helper
    add     sp, sp, #8
    ret
```

### 11.1.4 Correct Fix (Align Then Call)

```
/* GOOD: allocate multiple of 16 */
.text
.global fix_aligned_call
.type   fix_aligned_call, %function
fix_aligned_call:
```

```
    sub     sp, sp, #16
    str     x30, [sp, #8]       /* save LR if non-leaf */
    bl      helper
    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

### 11.1.5 Rule

- If you adjust `SP`, do it in multiples of **16**.
- Before every `bl`, ensure `SP % 16 == 0`.

## 11.2 Incorrect Register Preservation

### 11.2.1 The Failure

If your function modifies a callee-saved register and does not restore it, you corrupt the caller's state. Compilers rely heavily on callee-saved registers to keep values alive across calls.

### 11.2.2 Symptoms

- wrong results in the caller after returning,
- corruption that appears only with optimization,
- failures that move when you change unrelated code.

### 11.2.3 Bad Example (Clobber Preserved Register)

```
/* BAD: clobbers x19 without saving/restoring */
```

```
.text
.global bug_clobber_preserved
.type   bug_clobber_preserved, %function
bug_clobber_preserved:
    mov     x19, #123           /* ABI violation */
    add     x0, x0, #1
    ret
```

### 11.2.4 Correct Fix (Save/Restore)

```
/* GOOD: save/restore x19, keep SP aligned */
.text
.global fix_preserved_x19
.type   fix_preserved_x19, %function
fix_preserved_x19:
    sub     sp, sp, #16
    str     x19, [sp, #0]
    mov     x19, #123
    add     x0, x0, #1
    ldr     x19, [sp, #0]
    add     sp, sp, #16
    ret
```

### 11.2.5 Rule

- If you touch a callee-saved register, save it once in the prologue and restore once in the epilogue.

- Never early-return without restoring preserved state.

## 11.3 Mismatched Function Signatures

### 11.3.1 The Failure

The assembler does not know your C/C++ prototype. If your assembly function's assumed argument types/order do not match the actual C/C++ declaration, you will read the wrong registers or the wrong register bank.

### 11.3.2 The Most Common Mismatches

- treating a `double` as if it were in `xN` (it is in `dN`),
- mixing up argument order,
- using `int` vs `long` (size/class differences),
- returning FP in `x0` instead of `d0` (or vice versa).

### 11.3.3 Bad Example (Reads FP Argument From the Wrong Bank)

```
/* C/C++ expects: double f(double x); */
extern "C" double f(double x);
```

```
/* BAD: assumes x0 holds the double argument; it is in d0 */
.text
.global f
.type   f, %function
f:
    /* WRONG: x0 is not the FP argument here */
    add     x0, x0, #1
    ret
```

### 11.3.4 Correct Fix

```
/* GOOD: use d0 for double input/output */
.text
.global f
.type   f, %function
f:
    fadd    d0, d0, d0
    ret
```

### 11.3.5 Rule

- Freeze prototypes in a shared header.
- Keep assembly boundaries scalar/pointer-only when possible.
- Validate by comparing compiler-generated code for the same signature.

## 11.4 Variadic Function Pitfalls

### 11.4.1 The Failure

Variadic functions (`...`) depend on ABI-defined **va_list** and register save areas. Implementing a variadic callee in assembly without fully matching the platform's `va_list` model is a classic source of silent corruption.

### 11.4.2 Symptoms

- wrong values retrieved from `va_arg`,
- failures only when mixing integer and FP variadic arguments,

- behavior that changes with optimization or with different callers.

### 11.4.3 Bad Pattern (Assembly Variadic Callee Without Proper va_list)

```
/* BAD IDEA: variadic callee in assembly without implementing the ABI
 ↪  va_list model */
.text
.global bad_variadic
.type   bad_variadic, %function
bad_variadic:
    /* placeholder: any attempt to walk extra args here is
     ↪  ABI-sensitive and usually wrong */
    ret
```

### 11.4.4 Correct Strategy: Keep Variadic in C, Expose Fixed Wrapper

```
/* C/C++: variadic stays in C; wrapper is fixed and assembly-friendly */
#include <cstdint>
#include <cstdarg>

static std::uint64_t sum_va(int n, ...) {
    std::va_list ap;
    va_start(ap, n);
    std::uint64_t s = 0;
    for (int i = 0; i < n; ++i) {
        s += va_arg(ap, std::uint64_t);
    }
    va_end(ap);
    return s;
}
```

```
extern "C" std::uint64_t sum3_u64(std::uint64_t a, std::uint64_t b,
↪  std::uint64_t c) {
    return sum_va(3, a, b, c);
}
```

```
/* Assembly-friendly fixed boundary */
.text
.global sum3_u64_asm
.type   sum3_u64_asm, %function
sum3_u64_asm:
    add     x0, x0, x1
    add     x0, x0, x2
    ret
```

### 11.4.5 Rule

- Do not export variadic ABIs across modules as a “stable boundary”.
- Keep variadics inside one compilation domain and wrap them with fixed-arity functions.

## 11.5 Mixing ABIs Across Compilation Units

### 11.5.1 The Failure

Even on the same ISA, you can accidentally mix incompatible ABI assumptions across translation units:

- different language linkage (`extern "C"` missing),
- different structure packing/alignment settings,
- different floating-point ABI modes (platform-dependent),

- different calling convention attributes (toolchain-specific),
- mixing C and C++ without consistent headers.

### 11.5.2 Symptoms

- link succeeds but runtime values are wrong,
- only certain call sites fail,
- struct fields appear swapped or shifted,
- crashes when returning aggregates or passing structs.

### 11.5.3 Example: Packing Mismatch (Classic Silent Corruption)

```
/* TU1: packed layout */
#include <cstdint>
#pragma pack(push, 1)
struct S { std::uint8_t a; std::uint64_t b; };
#pragma pack(pop)
extern "C" std::uint64_t get_b(S s);
```

```
/* TU2: default layout (NOT packed) */
#include <cstdint>
struct S { std::uint8_t a; std::uint64_t b; };
extern "C" std::uint64_t get_b(S s); /* same name, different ABI layout:
↪  disaster */
```

This can produce silent corruption because the caller and callee disagree on:

- the size of `S`,
- the alignment of `b`,
- and therefore how the argument is passed (registers vs stack) and where its bytes reside.

### 11.5.4 Robust Fix Strategy

- Put **all interop declarations** in a single shared header.
- Avoid passing structs by value across boundaries; prefer pointers and explicit sizes.
- Avoid build-flag-dependent ABI in public interfaces.
- Prefer scalar-only and pointer-only ABI boundaries for long-term compatibility.

## 11.6 Practical ABI Breakage Checklist

- Every `bl` site: `SP` aligned to 16 bytes.
- Every assembly function: callee-saved registers you touch are saved/restored.
- Every boundary: signature matches exactly (types, order, return type).
- No variadic functions as exported boundaries; wrap with fixed-arity functions.
- No ABI-affecting mismatches across units (packing, linkage, calling convention attributes).

# Chapter 12

# Practical ABI Discipline Checklist

## 12.1 Mandatory Rules Recap

This chapter is a **non-negotiable** checklist for AAPCS64 correctness. If any item is violated, your code may work accidentally and then fail under optimization, different toolchains, or unrelated changes.

### 12.1.1 Call Boundary Laws (Always True)

- **Integer/pointer args:** `x0--x7` carry the first arguments.
- **FP args:** `v0--v7` carry the first FP/SIMD arguments (`s0/d0` views for scalars).
- **Integer/pointer return:** `x0`.
- **FP return:** `v0` (`s0`/`d0`).
- **SP alignment:** `SP` is **16-byte aligned** at every public boundary and at every `bl`.
- **No red-zone reliance:** do not read/write below `SP` unless you allocate it.

- **Callee-saved preservation:** if you touch a preserved register, you restore it before `ret`.
- **Caller-saved reality:** assume volatile regs are clobbered by calls; preserve your own live values.

### 12.1.2 One-Line Failure Model

If you cannot prove your function preserves the ABI invariants, it is not ABI-correct.

## 12.2 Safe Assembly–C/C++ Interoperability Checklist

### 12.2.1 A. Signature Discipline

- Keep the C/C++ prototype in a shared header used by **all** translation units.
- Use `extern "C"` for any function called from assembly or C.
- Avoid passing structs by value across assembly boundaries; prefer scalars and pointers.
- Avoid returning structs by value across assembly boundaries; prefer `out` pointers or scalar returns.

### 12.2.2 B. Stack Discipline

- Allocate stack in multiples of **16**.
- Before any `bl`, ensure `SP % 16 == 0`.
- Restore `SP` exactly before `ret`.

### 12.2.3 C. Register Discipline

- **Callee-saved:** save/restore any preserved register you modify.
- **Caller-saved:** save your live values before calls (stack or preserved regs).
- Save `LR` (`x30`) if you are non-leaf and need to return after calling.

### 12.2.4 D. C++ Runtime Discipline

- Do not let exceptions cross an assembly/C boundary.
- Do not call constructors/destructors directly from assembly as a public interface; wrap them.
- Use opaque handles (`void*`) for C++ objects in stable ABIs.

### 12.2.5 Example 1: ABI-Safe "Assembly Function Called from C" Template

```
/* Leaf, scalar-only, ABI-safe */
.text
.global asm_leaf_u64
.type   asm_leaf_u64, %function
asm_leaf_u64:
    /* x0 = arg0 */
    add     x0, x0, #1
    ret
```

### 12.2.6 Example 2: ABI-Safe "Assembly Calls C" Template (Non-Leaf)

```
/* Non-leaf: saves LR, keeps SP aligned, preserves live values if
 ↪  needed */
.text
.global asm_calls_c_template
.type   asm_calls_c_template, %function
asm_calls_c_template:
    sub     sp, sp, #16
    str     x30, [sp, #8]

    /* optional: preserve live caller-saved values before bl */
    /* str x0, [sp, #0] */

    bl      c_function

    /* optional: restore preserved live values */
    /* ldr x0, [sp, #0] */

    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

### 12.2.7 Example 3: ABI-Safe Mixed Integer + FP Boundary

```
/* C/C++ */
#include <cstdint>
extern "C" double api_mix(std::uint64_t a, double x, std::uint64_t b,
 ↪  double y);
```

```
/* a->x0, x->d0, b->x1, y->d1, return d0 */
```

```
.text
.global asm_forward_api_mix
.type   asm_forward_api_mix, %function
asm_forward_api_mix:
    sub     sp, sp, #16
    str     x30, [sp, #8]
    bl      api_mix
    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

## 12.3 ABI Validation Before Optimization

Optimization does not create ABI bugs; it **exposes** them. Validate ABI correctness before turning performance work into a moving target.

### 12.3.1 Validation Order (Practical)

1. **Freeze prototypes:** one shared header; `extern "C"` where required.
2. **Force real boundaries:** compile in separate translation units so calls remain calls.
3. **Test at `-O0` and `-O2`:** if behavior changes, suspect ABI violations.
4. **Audit stack:** confirm 16-byte alignment at every `bl`.
5. **Audit preserved regs:** list every callee-saved register used and verify save/restore.
6. **Compare with compiler output:** implement the same function in C and compare the ABI-visible behavior.

### 12.3.2 Example 4: "ABI Probe" Pattern Using a Known C Callee

A simple way to detect stack/reg problems is to call a known function and then use values after it. If your ABI discipline is wrong, corruption appears quickly.

```
/* Calls a known function; checks that our preserved state remains
 ↪  valid */
.text
.global abi_probe
.type   abi_probe, %function
abi_probe:
    sub     sp, sp, #32
    stp     x19, x20, [sp, #0]      /* preserve regs we will rely on
     ↪  */
    str     x30, [sp, #24]          /* save LR */

    mov     x19, x0                  /* keep a live value in
     ↪  preserved reg */
    bl      known_c_function          /* must not corrupt x19
     ↪  (callee-saved from our view) */
    add     x0, x19, #1               /* if x19 corrupted -> visible
     ↪  failure */

    ldr     x30, [sp, #24]
    ldp     x19, x20, [sp, #0]
    add     sp, sp, #32
    ret
```

## 12.4 When to Re-Read the Specification

Re-read the specification whenever you change anything that could alter ABI classification or call-boundary behavior.

### 12.4.1 Immediate Triggers

- You add or remove an argument, or change a type (`int` vs `long`, pointer vs integer, `float` vs `double`).
- You introduce aggregates (structs/unions) or return them by value.
- You add `...` (variadic arguments) or introduce `va_list`.
- You change packing/alignment pragmas or compiler flags that affect layout.
- You move code across compilation units or mix toolchains.
- You add a function call inside assembly (turning leaf into non-leaf).
- You start using SIMD registers as long-lived locals across calls.

### 12.4.2 Rule of Thumb

Re-read the ABI spec when the signature, the frame, or the toolchain changes.

### 12.4.3 Final Discipline Statement

- ABI correctness is a **proof obligation**.
- Performance comes **after** correctness.

- The simplest stable boundary is: **scalars + pointers + explicit ownership + no exceptions**.

# Appendices

## Appendix A — Minimal AAPCS64 Reference (Conceptual)

This appendix provides a concise, audit-oriented reference for AAPCS64. It is intended for last-minute ABI verification when writing or reviewing assembly that interoperates with C/C++.

### Register Usage Summary

#### General-Purpose Registers (Conceptual Roles)

- **x0–x7**: Integer / pointer argument registers (caller-saved)
- **x0**: Primary integer / pointer return register
- **x8**: Indirect result location register (hidden structure return)
- **x9–x15**: Temporary registers (caller-saved)
- **x16–x17**: Intra-procedure call temporaries (caller-saved; linker veneers)
- **x18**: Platform register (treat as caller-saved unless platform specifies otherwise)
- **x19–x28**: Callee-saved registers (must be preserved by callee)

- **x29**: Frame Pointer (FP) when used (callee-saved)
- **x30**: Link Register (LR)

### SIMD / Floating-Point Registers

- **v0–v7**: FP / SIMD argument registers (caller-saved)
- **v0**: FP return register (`s0` for `float`, `d0` for `double`)
- **v8–v15**: Callee-saved SIMD registers
- **v16–v31**: Temporary SIMD registers (caller-saved)

### Special Registers

- **SP**: Stack Pointer (must remain 16-byte aligned at call boundaries)
- **XZR**: Zero register (reads as zero, writes are discarded)

## Argument and Return Rules Summary

### Integer and Pointer Arguments

- Arguments 1–8: `x0--x7`
- Additional arguments: passed on the stack
- Pointer types follow integer rules

### Floating-Point Arguments

- FP scalar arguments (`float`, `double`): `v0--v7` using `sN`/`dN`
- Additional FP arguments: passed on the stack

### Return Values

- Integer / pointer return: `x0`
- 32-bit integer return: `w0`
- `float` return: `s0`
- `double` return: `d0`
- Small aggregates: may return in registers (ABI classification dependent)
- Large aggregates: returned indirectly via hidden pointer (commonly `x8`)

### Scalar Boundary Example

```
/* x0 = arg0, x1 = arg1, return in x0 */
.text
.global ref_add
.type   ref_add, %function
ref_add:
    add     x0, x0, x1
    ret
```

### Floating-Point Boundary Example

```
/* d0 = arg0, d1 = arg1, return in d0 */
.text
.global ref_fadd
.type   ref_fadd, %function
ref_fadd:
    fadd    d0, d0, d1
    ret
```

## Stack Alignment Rules Summary

### Core Stack Rules

- Stack grows toward lower addresses
- **SP must be 16-byte aligned**:
  - on function entry
  - before every `bl`
  - at function return
- Stack allocation must be in multiples of 16 bytes
- No red zone: memory below `SP` is not safe unless explicitly allocated

### Minimal Aligned Stack Allocation

```
/* ABI-correct stack allocation */
.text
.global ref_stack_alloc
.type   ref_stack_alloc, %function
ref_stack_alloc:
    sub     sp, sp, #16
    /* use [sp, #0..15] */
    add     sp, sp, #16
    ret
```

### Common Alignment Violation

```
/* BAD: misaligned SP */
.text
```

```
.global ref_bad_align
.type   ref_bad_align, %function
ref_bad_align:
    sub     sp, sp, #8          /* ABI violation */
    add     sp, sp, #8
    ret
```

### Canonical Non-Leaf Frame

```
/* Canonical FP/LR frame */
.text
.global ref_frame
.type   ref_frame, %function
ref_frame:
    sub     sp, sp, #16
    stp     x29, x30, [sp, #0]
    mov     x29, sp

    /* body */

    ldp     x29, x30, [sp, #0]
    add     sp, sp, #16
    ret
```

# Appendix B — Cross-Architecture Comparison (Conceptual)

This appendix provides a conceptual comparison between AAPCS64 and the dominant x86-64 ABIs. The goal is not memorization, but understanding **why ABI rules differ** and how architecture shapes calling conventions.

## AAPCS64 vs x86-64 System V ABI

### Register-Based Argument Passing

- **AAPCS64:**
  - Integer / pointer arguments: `x0--x7`
  - FP arguments: `v0--v7`
  - Clean separation between integer and FP register banks
- **x86-64 System V ABI:**
  - Integer / pointer arguments: `RDI, RSI, RDX, RCX, R8, R9`
  - FP arguments: `XMM0--XMM7`
  - Mixed integer and FP arguments share a unified calling sequence

### Return Value Rules

- **AAPCS64:**
  - Integer / pointer return: `x0`
  - FP return: `v0` (`s0`/`d0`)
  - Indirect (large struct) return commonly via hidden pointer in `x8`
- **x86-64 System V ABI:**
  - Integer / pointer return: `RAX`
  - FP return: `XMM0`
  - Large struct returns via hidden pointer in `RDI`

### Stack Alignment and Red Zone

- **AAPCS64:**
  - Stack must be 16-byte aligned at all call boundaries
  - No red zone; memory below `SP` is unsafe unless allocated
- **x86-64 System V ABI:**
  - Stack must be 16-byte aligned at call sites
  - 128-byte red zone exists below `RSP` for leaf functions

### Conceptual Example: Leaf Function

```
/* AAPCS64: leaf function, no red zone */
.text
.global aarch64_leaf
.type   aarch64_leaf, %function
aarch64_leaf:
    add     x0, x0, #1
    ret
```

```
/* x86-64 SysV: leaf function may use red zone */
.text
.global sysv_leaf
.type   sysv_leaf, %function
sysv_leaf:
    add     rax, rdi, 1
    ret
```

# AAPCS64 vs Windows x64 ABI

### Argument Registers

- **AAPCS64:**
  - Integer / pointer args: `x0--x7`
  - FP args: `v0--v7`
- **Windows x64 ABI:**
  - Integer / pointer args: `RCX, RDX, R8, R9`
  - FP args: `XMM0--XMM3`
  - Remaining arguments always passed on the stack

### Shadow Space vs Explicit Stack Discipline

- **AAPCS64:**
  - No mandatory shadow space
  - Stack space allocated only when needed
- **Windows x64 ABI:**
  - Mandatory 32-byte shadow space reserved by the caller
  - Callee may spill argument registers into this space

### Register Preservation Model

- **AAPCS64:**
  - Callee-saved: `x19--x28`, `x29`

  - Clear separation of volatile vs preserved registers

- **Windows x64 ABI:**

  - Larger set of callee-saved registers
  - Strong emphasis on predictable stack frames for debugging and unwinding

**Conceptual Example: Call Preparation**

```
/* AAPCS64: no mandatory outgoing stack space */
.text
.global aarch64_call
.type   aarch64_call, %function
aarch64_call:
    sub     sp, sp, #16
    str     x30, [sp, #8]
    bl      callee
    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

```
/* Windows x64: caller allocates shadow space */
.text
.global win64_call
.type   win64_call, %function
win64_call:
    sub     rsp, rsp, 32
    call    callee
    add     rsp, rsp, 32
    ret
```

## Architectural Reasons for Differences

### RISC vs CISC Design Philosophy

- AArch64 follows a **RISC** philosophy:
  - large uniform register file
  - explicit load/store model
  - simple, regular calling convention rules
- x86-64 evolved from a **CISC** lineage:
  - fewer architectural registers historically
  - strong backward compatibility constraints
  - ABI designs that accommodate legacy behavior

### Register File Size and ABI Design

- AArch64 has 31 general-purpose registers, enabling:
  - more arguments in registers
  - reduced stack traffic
  - simpler ABI rules
- x86-64 has fewer architectural GPRs, leading to:
  - earlier stack spilling
  - ABI features like red zones or shadow space

**Operating System and Toolchain Influence**

- Windows prioritizes:
  - consistent stack layout
  - structured exception handling
  - debugger-friendly unwinding
- Unix-like systems prioritize:
  - performance
  - leaf-function optimization
  - flexible compiler optimization strategies

**Final Conceptual Takeaway**

- ABI rules are not arbitrary; they reflect architectural and OS design goals.
- AAPCS64 emphasizes simplicity, regularity, and performance.
- x86-64 ABIs reflect historical constraints and OS-specific priorities.
- Understanding the **reasoning** behind ABIs makes cross-architecture work predictable and safe.

# Appendix C — Preparation for Advanced Topics

This appendix defines the **readiness criteria** required before moving beyond basic AAPCS64 interoperability. Each subsection identifies what must already be correct at the ABI level before advanced mechanisms are safe to introduce.

## Readiness for Exception Unwinding

### Why Exception Unwinding Is ABI-Sensitive

Exception unwinding depends on:

- precise stack frame construction,
- correct save/restore of callee-saved registers,
- consistent use (or non-use) of frame pointers,
- accurate call-return linkage.

If any ABI rule is violated, the unwinder may:

- skip frames,
- restore incorrect register state,
- or terminate execution during unwinding.

### Minimum ABI Requirements Before Enabling Unwinding

- Every non-leaf function saves and restores `x30` correctly.
- Any used callee-saved register (`x19--x28`, `x29`) is preserved.
- `SP` is restored exactly to its entry value on all exit paths.
- Stack frames follow a consistent pattern suitable for unwinding.

### Canonical Unwind-Friendly Frame

```
/* Frame suitable for unwinding and debugging */
.text
.global unwind_ready_fn
.type   unwind_ready_fn, %function
unwind_ready_fn:
    sub     sp, sp, #16
    stp     x29, x30, [sp, #0]
    mov     x29, sp

    /* body */

    ldp     x29, x30, [sp, #0]
    add     sp, sp, #16
    ret
```

### Rule of Readiness

> If your function cannot be unwound safely, it is not ready for exceptions.

## Readiness for JIT and FFI Systems

### Why JIT/FFI Environments Amplify ABI Errors

JIT and FFI systems:

- generate calls dynamically,
- cannot rely on compile-time verification,
- and often bridge multiple languages and runtimes.

Any ABI ambiguity becomes a runtime failure that is difficult to diagnose.

### Mandatory ABI Constraints for JIT/FFI

- Only use **documented ABI-visible registers**.
- Do not rely on compiler-specific register allocation behavior.
- Keep boundaries strictly scalar/pointer-based.
- Do not expose variadic or template-heavy interfaces.
- Ensure stack alignment before every generated call.

### FFI-Safe Function Shape

```
/* FFI-friendly: scalar args, scalar return, no hidden behavior */
.text
.global ffi_safe_add
.type   ffi_safe_add, %function
ffi_safe_add:
    /* x0 = a, x1 = b */
    add     x0, x0, x1
    ret
```

### JIT Call Stub (Conceptual)

```
/* JIT-generated call stub must obey ABI rules */
.text
.global jit_call_stub
.type   jit_call_stub, %function
jit_call_stub:
```

```
    sub     sp, sp, #16
    str     x30, [sp, #8]

    /* x0..xN populated by JIT runtime */
    bl      target_function

    ldr     x30, [sp, #8]
    add     sp, sp, #16
    ret
```

#### Rule of Readiness

> If your ABI contract is not explicit, a JIT or FFI will eventually break it.

## Readiness for OS Kernel Boundaries

### Why Kernel Boundaries Are Different

Crossing into kernel mode introduces:

- privilege level changes,
- strict entry/exit conventions,
- architectural state preservation requirements,
- and security constraints.

User-space ABI mistakes that appear benign can become fatal at kernel boundaries.

### Minimum Discipline Before Kernel Interfaces

- Do not assume user-space calling conventions extend into kernel internals.

- Preserve only what the kernel ABI explicitly requires.
- Never leak user-space stack assumptions across the boundary.
- Treat kernel calls as strict ABI black boxes.

### User-to-Kernel Call Shape (Conceptual)

```
/* User-space syscall-style boundary (conceptual) */
.text
.global user_kernel_call
.type   user_kernel_call, %function
user_kernel_call:
    /* x0..xN contain arguments */
    /* transition instruction omitted (conceptual) */
    ret
```

### Kernel-Safe Assembly Discipline

- Never assume kernel preserves user registers beyond its contract.
- Never assume user `SP` survives unchanged.
- Validate all pointers before passing across the boundary.
- Treat the boundary as hostile and minimal.

### Rule of Readiness

If your code is sloppy with ABI rules, it has no place near the kernel.

## Final Readiness Summary

- Exception unwinding requires perfectly disciplined frames.
- JIT and FFI require explicit, minimal, and verifiable ABI contracts.
- Kernel boundaries require absolute correctness and zero assumptions.
- Advanced topics magnify ABI mistakes; they do not tolerate them.

  Mastery of the ABI is not optional preparation for advanced systems work—it is the prerequisite.

# References

This chapter summarizes the **authoritative conceptual sources** that underpin this booklet. It intentionally focuses on **categories of trusted material** rather than URLs, ensuring long-term validity and academic clarity.

## ARM Architecture Conceptual Manuals

The conceptual foundation of AAPCS64 rests on ARM's architectural documentation, which defines:

- the AArch64 execution state,
- general-purpose and SIMD/FP register files,
- privilege levels and exception models,
- instruction execution semantics independent of operating systems.

These manuals are essential for understanding:

- why argument registers are plentiful in AArch64,
- why stack alignment rules are strict,
- how architectural state is preserved across calls and exceptions.

Within this booklet, ARM architecture manuals are used:

- conceptually (not as instruction listings),
- to justify ABI design choices,
- to explain architectural constraints that shape calling conventions.

# AAPCS64 Specification (Conceptual Use)

The AAPCS64 specification defines the **contract between separately compiled units**. It specifies:

- argument classification and placement,
- return value rules,
- callee-saved vs caller-saved registers,
- stack alignment and frame discipline,
- indirect passing and return mechanisms.

In this booklet, the specification is used:

- as a source of **invariants**, not recipes,
- to explain what must always be true at call boundaries,
- to derive safe assembly–C/C++ interoperability patterns.

The emphasis is on:

- stable ABI behavior,
- cross-toolchain correctness,
- long-term maintainability rather than compiler-specific quirks.

# Compiler ABI Documentation

Modern compilers implement AAPCS64 while retaining freedom in:

- prologue/epilogue generation,
- register allocation,
- frame pointer usage,
- optimization strategies.

Compiler ABI documentation is used in this booklet to clarify:

- what compilers must guarantee versus what they may change,
- how optimization interacts with ABI preservation,
- why inspecting compiler-generated code is a valid validation strategy.

The guiding principle applied throughout:

> If the ABI does not guarantee it, your code must not rely on it.

# Cross-References to Other Booklets in This Series

This booklet is part of a structured CPU Programming Series. Its content builds directly on concepts introduced earlier and prepares the ground for later topics.

## Foundational Dependencies

Readers are expected to be comfortable with:

- instruction execution models,

- register semantics and flag behavior,
- stack mechanics and call/return flow.

These are covered in earlier booklets focusing on:

- CPU execution pipelines,
- registers and binary data representation,
- stack discipline and calling conventions at a conceptual level.

## Forward Connections

The ABI discipline established here is a prerequisite for:

- exception unwinding and runtime support,
- foreign-function interfaces and JIT systems,
- OS kernel boundaries and syscall interfaces,
- mixed-language and multi-architecture systems.

Later booklets in the series assume:

- strict adherence to AAPCS64 rules,
- confidence in reading compiler-generated assembly,
- and the ability to reason about call boundaries without guesswork.

# Final Reference Note

This booklet intentionally avoids transient sources. Its reference model is based on:

- architectural specifications,
- formal ABI definitions,
- and compiler contracts that evolve slowly and predictably.

  Mastering the ABI is not about memorizing rules—it is about understanding the guarantees that make large systems possible.