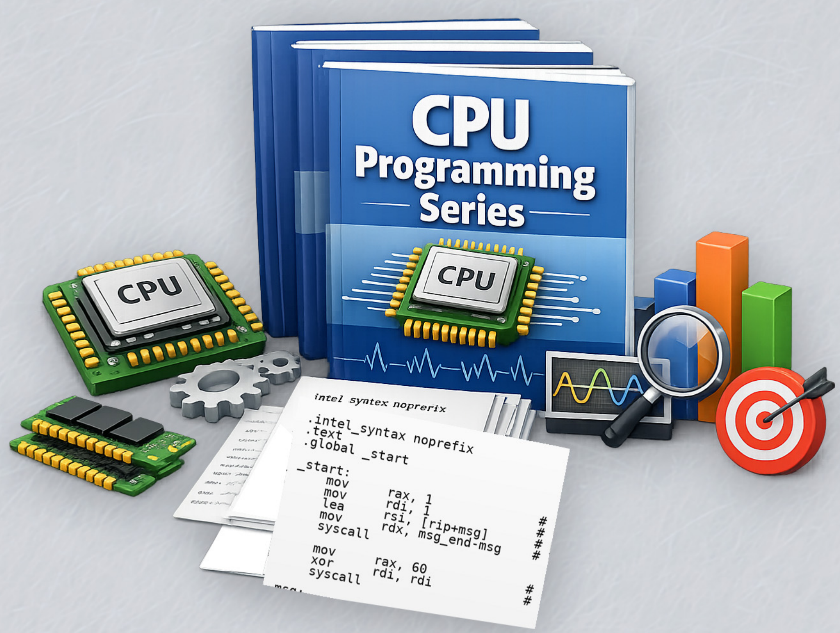
https://simplifycpp.org

CPU Programming Series

x86-64 System V ABI

Calling Convention & Stack Discipline



```
.intel_syntax noprefix
.text
.global _start
_start:
    mov     rax, 1
    mov     rdi, 1
    lea     rsi, [rip+msg]
    mov     rdx, msg_end-msg
    syscall
    mov     rax, 60
    xor     rdi, rdi
    syscall
```

9

Prepared by Ayman Alheraki

CPU Programming Series

x86-64 System V ABI

Calling Convention & Stack Discipline

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Purpose of This Booklet

This booklet is dedicated to a precise and practical understanding of the **x86-64 System V ABI**, which is the dominant user-space binary interface on Linux, BSD, macOS-derived systems, and many Unix-like environments. Its primary purpose is to explain how calling conventions and stack discipline are formally defined at the ABI level, and why strict adherence to these rules is mandatory for correct execution.

The System V ABI is often learned implicitly through compiler output or online examples, yet many subtle rules are either misunderstood or incorrectly assumed to be identical across platforms. This booklet exists to eliminate those assumptions by presenting the ABI as a *binary contract*, not as a collection of compiler habits.

The focus is on:

- Correct argument passing
- Stack layout and alignment
- Register preservation rules
- Call and return mechanics

Every rule explained here directly impacts correctness, interoperability, and long-term maintainability of low-level code.

Scope and Assumptions

This booklet focuses exclusively on the **System V AMD64 ABI** as used in modern 64-bit user-space programs. It covers the ABI guarantees that apply across compliant compilers and operating systems, independent of optimization level or language frontend.
The scope includes:

- General-purpose and vector register roles
- Integer and floating-point argument passing
- Stack frame structure and red zone usage
- Stack alignment requirements at call boundaries
- Caller-saved and callee-saved register responsibilities

The following assumptions are made:

- The reader understands basic x86-64 architecture concepts
- The reader is familiar with registers, memory addressing, and function calls
- The reader may be working in C, C++, or handwritten assembly

This booklet does not attempt to teach assembly language syntax from zero, nor does it cover operating system internals beyond what is required to understand ABI behavior.

How to Read and Use This Booklet Effectively

This booklet is designed to be read sequentially. Each chapter builds upon invariants established earlier, especially regarding stack alignment and register volatility. Skipping foundational chapters often leads to incorrect mental models of later examples.
All assembly examples:

- Use Intel syntax
- Follow the System V AMD64 ABI strictly
- Are intentionally minimal to expose ABI rules clearly

Incorrect examples are shown only to demonstrate why code breaks when ABI rules are violated. They should never be treated as acceptable shortcuts.

```
# Minimal System V AMD64 function prologue (illustrative)
# RSP must be 16-byte aligned at the call boundary.
push rbp
mov  rbp, rsp
```

When using this booklet:

- Track register ownership mentally at every call boundary
- Verify stack alignment before every call instruction
- Treat all ABI rules as mandatory, not advisory
- Revisit this booklet when writing assembly, inline assembly, or FFI code

This booklet is intended to function both as a structured learning path and as a long-term reference. Mastery of the System V ABI comes from disciplined practice and repeated validation against real code.

Chapter 1

Introduction to the System V AMD64 ABI

1.1 What an ABI Defines and Guarantees

An **Application Binary Interface (ABI)** is the machine-level contract that allows independently compiled binaries to work together correctly at run time. If two components follow the same ABI, they can interoperate even if they were built by different compilers, in different languages, or at different times.
For the **System V AMD64 ABI**, the ABI defines and guarantees (at minimum):

- **Calling convention:** where arguments are placed (registers vs stack), how return values are delivered, and who is responsible for preserving which CPU state.
- **Register roles:** volatile (caller-saved) vs non-volatile (callee-saved) registers.
- **Stack discipline:** stack growth direction, stack frame conventions, and **16-byte alignment rules** at call boundaries.
- **Binary interface data rules:** size/alignment of fundamental types, struct/union layout rules, and how aggregates are passed/returned at the binary level.

- **Object format and relocation model:** how code/data are represented in object files and how the linker/loader resolves addresses.
- **Language linkage conventions:** for example, C vs C++ name mangling, enabling stable linking via `extern "C"` boundaries.

In this booklet, the ABI areas that most commonly cause correctness failures are: **argument passing**, **register preservation**, and **stack alignment**.

Example: "Works in debug, breaks in release" is often an ABI failure

A typical failure mode is incorrect stack alignment. Many functions appear to work until a call chain introduces vector instructions or stack-based spills that assume ABI-mandated alignment.

```
.intel_syntax noprefix
.global bad_alignment_sysv
bad_alignment_sysv:
    # # BUG: breaks SysV call-site alignment if we call another
     ↪  function.
    # # (After 'call', the callee begins with RSP misaligned by 8
     ↪  relative to 16.)
    push rbx
    call external_func     # # may crash or misbehave if it assumes
     ↪  ABI alignment
    pop  rbx
    ret
```

```
.intel_syntax noprefix
.global fixed_alignment_sysv
fixed_alignment_sysv:
    push rbx
    sub  rsp, 8             # # restore 16-byte alignment before call
    call external_func
    add  rsp, 8
    pop  rbx
    ret
```

The key point: the ABI is not "style." It is a correctness requirement that can be tested by real code paths.

1.2 Role of the ABI in Compiled Programs

A compiled program is a composition of separately built components:

- Your translation units compiled to object files
- Static libraries and shared libraries
- Runtime libraries (C runtime, C++ runtime, unwinding/runtime support)
- The dynamic loader and OS-provided modules

The ABI is the glue that makes these components function as a single coherent executable system. It enables:

- **Reliable function calls** across module boundaries
- **Stable linkage** against system libraries and third-party libraries
- **Correct stack walking and unwinding** for debugging and exceptions
- **Foreign-function interfaces (FFI)** across languages that agree on the ABI

At the CPU level, this means every call boundary must preserve the invariants expected by any other ABI-compliant code:

- argument locations are deterministic,
- register volatility rules are respected,
- stack pointer state and alignment are correct.

Example: The ABI is what makes separate compilation safe

A caller compiled in one file and a callee compiled in another file will still interoperate because both sides implement the same ABI contract.

```
.intel_syntax noprefix
# Concept: caller and callee compiled separately still agree on:
# - first integer arg in RDI (SysV)
# - return value in RAX
#
# long inc(long x);

.global inc_sysv
inc_sysv:
    lea rax, [rdi + 1]
    ret
```

Any SysV-compliant caller can call `inc_sysv` without knowing how it was compiled, as long as both sides obey the ABI.

1.3 Relationship Between ABI, Compiler, and Operating System

The ABI lives at the boundary between:

- **Compiler responsibilities** (code generation)
- **Operating system responsibilities** (loading/execution environment)
- **Toolchain responsibilities** (assembler, linker, loader, runtime)

1.3.1 ABI and the compiler

The compiler must emit code that respects the ABI:

- place integer/pointer arguments in `RDI, RSI, RDX, RCX, R8, R9`,
- place floating-point arguments in `XMM0--XMM7`,
- preserve callee-saved registers (`RBX, RBP, R12--R15`),
- maintain stack alignment at call boundaries,
- return values through `RAX` (integers/pointers) or `XMM0` (floating-point).

When you introduce handwritten assembly, inline assembly, JIT code, or unconventional FFI boundaries, you assume part of the compiler's job and must enforce ABI invariants yourself.

1.3.2 ABI and the operating system

The operating system and loader establish the execution environment in which ABI-compliant code runs:

- initial stack state at program entry,
- dynamic linking model and symbol resolution,
- thread startup conventions,
- signal/trap delivery mechanisms (separate from the function-call ABI).

The user-space ABI is what most code relies on for safe inter-module function calls; it is distinct from the system-call interface.

Example: user-space calling convention vs system call convention

Even on the same OS, the system-call mechanism may use different registers than the user-space ABI for normal function calls.

```
.intel_syntax noprefix
# Example only: user-space SysV function calls use RDI/RSI/RDX/...
# but an OS syscall interface can use a different register mapping.
# This distinction matters when writing low-level code near the
↪  boundary.
```

Practical Goal of This Booklet

This booklet turns the System V AMD64 ABI from “something you vaguely rely on” into a **disciplined mental model** you can apply when:

- reading compiler output,
- debugging stack corruption,
- writing correct assembly,

- building safe FFI boundaries,
- auditing code for ABI correctness.

The next chapters will formalize the register roles, stack layout, and alignment rules that must hold at every call boundary in System V AMD64 user-space code.

Chapter 2

General-Purpose Register Roles

2.1 Overview of x86-64 GPRs

In x86-64 long mode, the architectural **general-purpose registers (GPRs)** form a 16-register, 64-bit integer register file used for pointers, addresses, integer arithmetic, bit operations, loop/state variables, and general program state:

- **RAX, RBX, RCX, RDX**
- **RSI, RDI, RBP, RSP**
- **R8–R15**

Each register has subregister views:

- 64-bit: `RAX`
- 32-bit: `EAX`
- 16-bit: `AX`

- 8-bit: `AL` (low 8), and some legacy high-8 forms such as `AH`

A critical long-mode semantic used heavily by compilers:

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

Example: 32-bit write zero-extends to 64-bit

```
.intel_syntax noprefix
.global zero_extend_gpr_demo
zero_extend_gpr_demo:
    mov rax, -1          # # RAX = 0xFFFF_FFFF_FFFF_FFFF
    mov eax, 7           # # EAX write => RAX = 0x0000_0000_0000_0007
    ret
```

Two registers have special calling-sequence meaning in every ABI:

- **RSP** is the stack pointer and must remain valid; it must be restored before `ret`.
- **RBP** is a general register, but is often used as a frame pointer depending on compiler and debug settings.

2.2 Registers Used for Argument Passing (System V AMD64)

The System V AMD64 ABI defines register-based argument passing for integer/pointer types as follows:

- First six integer/pointer arguments: `RDI, RSI, RDX, RCX, R8, R9`
- Additional integer/pointer arguments: passed on the stack

Important properties for correctness:

- The integer/pointer register sequence is independent from the floating-point sequence (covered in the next chapter).
- Stack arguments are placed by the caller at higher addresses than the return address and are read by the callee relative to its stack/frame pointer strategy.
- The caller must keep **stack alignment correct at every call boundary** (details in the stack discipline chapters).

Example: Passing 1 to 8 integer arguments

```
.intel_syntax noprefix
.extern func8

# long func8(long a,long b,long c,long d,long e,long f,long g,long
 ↪  h);

.global call_func8_sysv
call_func8_sysv:
    # # First 6 args in registers
    mov rdi, 1          # # a
    mov rsi, 2          # # b
    mov rdx, 3          # # c
    mov rcx, 4          # # d
    mov r8,  5          # # e
    mov r9,  6          # # f

    # # Args 7 and 8 go on stack (right-to-left so that g is closest
     ↪  to return address)
    # # Reserve 16 bytes for g and h, and keep call-site alignment
     ↪  correct.
    sub rsp, 16
    mov qword ptr [rsp + 8], 8    # # h (8th)
    mov qword ptr [rsp + 0], 7    # # g (7th)

    call func8

    add rsp, 16
    ret
```

2.3 Registers Used for Return Values

System V AMD64 specifies that return values are delivered primarily through registers:

- **Integer/pointer return:** `RAX`
- **64-bit integer return:** `RAX`
- **Smaller integer returns:** placed in `AL/AX/EAX` and observed via ABI-defined extension rules according to the declared return type

A common additional rule used by many ABIs (and relied on by compilers):

- Some multi-word integer returns may use `RDX:RAX` as a pair, but scalar integer/pointer returns are always in `RAX`.

Example: returning a 64-bit integer

```
.intel_syntax noprefix
.global ret_const_i64
```

```
ret_const_i64:
    mov rax, 0x1122334455667788
    ret
```

Example: returning a boolean-like value

```
.intel_syntax noprefix
.global is_positive_sysv
# long is_positive(long x)  # x in RDI, return in RAX (0 or 1)
is_positive_sysv:
    xor eax, eax         # # default return 0
    test rdi, rdi
    setg al              # # AL=1 if x > 0 (signed)
    ret
```

2.4 Volatile and Non-Volatile Register Classification

The System V AMD64 ABI divides GPRs into two categories:

2.4.1 Non-volatile (callee-saved)

A callee must preserve these registers (restore original values before returning):

```
RBX, RBP, R12, R13, R14, R15
```

2.4.2 Volatile (caller-saved)

A caller must assume these registers may be clobbered by a call:

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

Special rule:

- **RSP** must be restored to its incoming value before `ret`. It is not a general “volatile” scratch register.

Example: Correct callee preservation of RBX

If a callee wants to use `RBX`, it must save and restore it.

```
.intel_syntax noprefix
.global use_rbx_sysv_correct
use_rbx_sysv_correct:
    push rbx             # # preserve non-volatile RBX
    mov  rbx, 123
    # # ... do work using RBX ...
    mov  eax, 0
    pop  rbx             # # restore RBX
    ret
```

Example: Caller saving a volatile register across a call

If the caller needs a value in `R10` after a function call, it must save it itself because `R10` is volatile.

```
.intel_syntax noprefix
.extern callee

.global caller_saves_r10_sysv
caller_saves_r10_sysv:
    mov r10, 0xABCDEF
    push r10             # # caller saves volatile register
    call callee
    pop  r10              # # restore after call
    mov  rax, r10
    ret
```

Example: ABI mismatch that breaks ported code

A common portability failure is assuming Windows x64 preservation rules on SysV. On SysV, `RDI` is volatile. If a caller wrongly assumes `RDI` survives a call, it may read garbage afterward.

```
.intel_syntax noprefix
.extern callee

.global wrong_assumption_rdi_sysv
wrong_assumption_rdi_sysv:
    mov rdi, 77
    call callee            # # callee may clobber RDI on SysV
    # # BUG: assuming RDI still holds 77
    mov rax, rdi
    ret
```

Discipline Summary

Correct System V AMD64 low-level code requires:

- Placing integer/pointer arguments in `RDI, RSI, RDX, RCX, R8, R9` in order.
- Reading return values from `RAX`.
- Treating `RBX, RBP, R12--R15` as callee-saved and preserving them when used.

- Treating `RAX, RCX, RDX, RSI, RDI, R8--R11` as caller-saved and saving them in the caller if needed across calls.

Chapter 3

Floating-Point and Vector Registers

3.1 XMM/YMM Register Purpose

In the System V AMD64 ABI, all floating-point and SIMD computation is performed using the **SSE and AVX register files**. The legacy x87 floating-point stack is not used for normal argument passing or return values in modern 64-bit user-space code.
The architectural vector registers are:

- **XMM0–XMM15**: 128-bit registers used for scalar floating-point values (`float`, `double`) and SSE vector operations.
- **YMM0–YMM15**: 256-bit registers introduced with AVX; each `YMMn` extends `XMMn` with an upper 128-bit lane.

Key architectural and ABI-relevant properties:

- Writing to `XMMn` updates only the lower 128 bits of `YMMn`.
- Writing to `YMMn` updates the full 256 bits.

- The ABI treats XMM/YMM registers as the carriers of both scalar FP and SIMD values.

Example: Partial vs full vector register writes

```
.intel_syntax noprefix
.global xmm_only_write
xmm_only_write:
    # Clears only the low 128 bits of YMM0
    xorps xmm0, xmm0
    ret
```

```
.intel_syntax noprefix
.global ymm_full_write
ymm_full_write:
    # Clears all 256 bits of YMM0
    vxorps ymm0, ymm0, ymm0
    ret
```

Relying on upper YMM state after an XMM-only write is a common low-level bug.

3.2 Floating-Point Argument Passing Rules

The System V AMD64 ABI defines a **separate argument register sequence** for floating-point values, independent of integer/pointer arguments.

3.2.1 Scalar floating-point arguments

- Up to **8** floating-point arguments are passed in `XMM0--XMM7`.
- Supported scalar types include `float`, `double`, and vector-compatible scalar values.

- Floating-point arguments do *not* consume integer argument registers.

If more than eight floating-point arguments are present, the remaining ones are passed on the stack according to ABI stack layout rules.

Example: Mixed integer and floating-point arguments

Conceptual signature:

```
.intel_syntax noprefix
# double f(long a, double b, long c, double d, double e);
```

System V register assignment:

- `a` in `RDI`
- `b` in `XMM0`
- `c` in `RSI`
- `d` in `XMM1`
- `e` in `XMM2`

```
.intel_syntax noprefix
.extern f

.global call_f_sysv
call_f_sysv:
    mov rdi, 10           # # a
    mov rsi, 20           # # c
    # # b, d, e loaded into XMM0, XMM1, XMM2 by caller
    call f
    ret
```

3.2.2 Floating-point return values

System V AMD64 specifies:

- **Scalar floating-point return:** `XMM0`
- **Vector return (SSE width):** `XMM0`

Example: Returning a double

```
.intel_syntax noprefix
.global ret_double_sysv
ret_double_sysv:
    # Return value must be in XMM0
    ret
```

3.3 SIMD Considerations in Function Calls

SIMD usage introduces strict ABI responsibilities related to register volatility, stack alignment, and call boundaries.

3.3.1 Vector register volatility (System V AMD64)

All vector registers are **caller-saved**:

- `XMM0--XMM15` are volatile
- `YMM0--YMM15` are volatile

A callee may freely overwrite any vector register without saving it.

Example: Caller responsibility for preserving XMM registers

```
.intel_syntax noprefix
.extern callee

.global caller_preserves_xmm0
caller_preserves_xmm0:
    sub  rsp, 16
    movdqu [rsp], xmm0      # # save volatile XMM0
    call callee
    movdqu xmm0, [rsp]      # # restore after call
    add  rsp, 16
    ret
```

3.3.2 Stack alignment and SIMD

The System V AMD64 ABI requires:

- **RSP must be 16-byte aligned at every call boundary**.

This requirement exists primarily to support aligned SIMD loads, spills, and ABI-compliant stack frames generated by compilers.

Example: Correct alignment before a call

```
.intel_syntax noprefix
.global aligned_call_sysv
aligned_call_sysv:
    push rbx                # # breaks alignment by 8
    sub  rsp, 8             # # restore 16-byte alignment
    call callee
    add  rsp, 8
    pop  rbx
    ret
```

Failing to maintain alignment may cause crashes, data corruption, or severe performance penalties once vector spills or aligned loads are introduced.

3.3.3 AVX and call-boundary hygiene

When AVX instructions are used, upper YMM state becomes live. Although the System V ABI allows unrestricted use of YMM registers, transitioning between AVX-heavy code and legacy SSE-only code can introduce penalties.
A disciplined practice at call boundaries is to clear upper YMM lanes when needed.

Example: Clearing upper YMM state before a call

```
.intel_syntax noprefix
.extern legacy_sse_func

.global avx_call_boundary_sysv
avx_call_boundary_sysv:
    # ... AVX work using YMM registers ...
    vzeroupper              # # clear upper 128 bits of all YMM regs
    call legacy_sse_func
    ret
```

Discipline Summary

- XMM/YMM registers are the sole carriers of scalar FP and SIMD values in System V AMD64.
- Up to eight floating-point arguments are passed in `XMM0--XMM7`, independent of integer arguments.
- All vector registers are caller-saved; callees may freely clobber them.
- Stack alignment at 16 bytes is mandatory for correct SIMD behavior.
- AVX/SSE transitions require explicit hygiene to avoid penalties and state hazards.

Chapter 4

Register-Based Argument Passing

4.1 Integer and Pointer Argument Rules

Under the **System V AMD64 ABI**, integer-class arguments (including pointers) are passed in a fixed register sequence. The first six integer/pointer arguments are placed in:

```
RDI, RSI, RDX, RCX, R8, R9
```

Additional integer/pointer arguments beyond the sixth are passed on the stack.
Important discipline rules:

- The register assignment is by **argument position among integer-class arguments**, not by type size.
- Narrow integer types (`char`, `short`, `int`) are carried in the corresponding 64-bit register, but the **caller must apply the correct extension** (sign or zero) to match the declared type rules at the call boundary.
- Pointers are passed as 64-bit values in the same integer-class registers.

- Stack-passed arguments are placed by the caller in memory above the return address (and any alignment padding), and are accessed by the callee relative to `RSP`/`RBP` according to its chosen frame strategy.

Example: Passing 1 to 6 integer/pointer arguments in registers

```
.intel_syntax noprefix
.extern f6

# long f6(long a, long b, long c, long d, long e, long f);

.global call_f6_sysv
call_f6_sysv:
    mov rdi, 1      # # a
    mov rsi, 2      # # b
    mov rdx, 3      # # c
    mov rcx, 4      # # d
    mov r8,  5      # # e
    mov r9,  6      # # f
    call f6
    ret
```

Example: Passing more than 6 integer arguments (stack extension)

```
.intel_syntax noprefix
.extern f8

# long f8(long a,long b,long c,long d,long e,long f,long g,long h);

.global call_f8_sysv
```

```
call_f8_sysv:
    mov rdi, 1       # # a
    mov rsi, 2       # # b
    mov rdx, 3       # # c
    mov rcx, 4       # # d
    mov r8,  5       # # e
    mov r9,  6       # # f

    # # Remaining args go on stack.
    # # Place right-to-left so that the 7th arg is closest to the
    ↪  return address.
    # # Reserve 16 bytes for g and h.
    sub rsp, 16
    mov qword ptr [rsp + 8], 8   # # h (8th)
    mov qword ptr [rsp + 0], 7   # # g (7th)

    call f8
    add rsp, 16
    ret
```

4.2 Floating-Point Argument Rules

Floating-point arguments use a separate register sequence:

```
XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7
```

Discipline rules:

- Up to **8** floating-point arguments are passed in `XMM0--XMM7`.

- Floating-point arguments do **not** consume integer argument registers.
- Additional floating-point arguments beyond the eighth are passed on the stack.
- Scalar `float` and `double` values are carried in XMM registers (low lanes), and callers must ensure correct value representation.

Example: Passing 1 to 8 double arguments in XMM registers

```
.intel_syntax noprefix
.extern fd8

# double fd8(double a,double b,double c,double d,double e,double
↪  f,double g,double h);

.global call_fd8_sysv
call_fd8_sysv:
    # # a..h must be placed in XMM0..XMM7 by the caller.
    call fd8
    ret
```

4.3 Mixed Argument Lists

In System V AMD64, mixed argument lists are handled by maintaining **two independent counters**:

- one counter for integer/pointer arguments (assigning `RDI, RSI, RDX, RCX, R8, R9`)
- one counter for floating-point arguments (assigning `XMM0--XMM7`)

This means:

- You can exhaust integer registers while still having available XMM registers.
- You can exhaust XMM registers while still having available integer registers.
- Stack placement happens independently once a class runs out of its register slots.

Example: Mixed signature with interleaving types

Conceptual signature:

```
.intel_syntax noprefix
# double mix(long a, double b, long c, double d, long e, double f);
```

Register assignment:

- `a` (int) in `RDI`
- `b` (fp) in `XMM0`
- `c` (int) in `RSI`
- `d` (fp) in `XMM1`
- `e` (int) in `RDX`
- `f` (fp) in `XMM2`

```
.intel_syntax noprefix
.extern mix

.global call_mix_sysv
call_mix_sysv:
    mov rdi, 11       # # a
    mov rsi, 22       # # c
    mov rdx, 33       # # e
    # # b in XMM0, d in XMM1, f in XMM2 (loaded elsewhere)
    call mix
    ret
```

4.4 Argument Classification and Ordering

The ABI performs **argument classification** to determine whether each argument is passed in an integer register, a vector register, or on the stack.
For practical assembly-level correctness in this booklet, treat the following as the core classification model:

- **INTEGER class:** integers, pointers, and values that are naturally represented as 64-bit integer quantities.
- **SSE class:** scalar floating-point values and vector values that are passed via XMM registers.
- **MEMORY class:** values that do not fit ABI register passing rules (commonly larger aggregates) and therefore are passed on the stack or via hidden pointers.

Ordering rules you must enforce at the call site:

- Arguments are **logically ordered left-to-right** by the function signature.
- Each argument is assigned to the next available register in its class sequence (INTEGER or SSE).
- Once a class exhausts its register slots, remaining arguments of that class are passed on the stack.

- Stack-passed arguments are laid out by the caller in increasing memory addresses such that the *earliest stack argument* is closest to the return address.

Example: Exhaust integer registers, continue using XMM registers

Conceptual signature:

```
.intel_syntax noprefix
# double g(long a,long b,long c,long d,long e,long f,long g,long h,
↪  double x, double y);
```

Register assignment:

- `a..f` in `RDI,RSI,RDX,RCX,R8,R9`
- `g,h` on stack (integer registers exhausted)
- `x,y` still in `XMM0,XMM1` (XMM registers remain available)

```
.intel_syntax noprefix
.extern g

.global call_g_sysv
call_g_sysv:
    mov rdi, 1
    mov rsi, 2
    mov rdx, 3
    mov rcx, 4
    mov r8,  5
    mov r9,  6

    sub rsp, 16
    mov qword ptr [rsp + 8], 8    # # h
    mov qword ptr [rsp + 0], 7    # # g

    # # x in XMM0, y in XMM1 (loaded elsewhere)
    call g

    add rsp, 16
    ret
```

Example: Exhaust XMM registers, continue using integer registers

Conceptual signature:

```
.intel_syntax noprefix
# long h(double a,double b,double c,double d,double e,double f,double
 ↪  g,double h,double i, long p);
```

Register assignment:

- `a..h` in `XMM0..XMM7`
- `i` on stack (XMM registers exhausted)
- `p` still in `RDI` (integer registers still available)

```
.intel_syntax noprefix
.extern h

.global call_h_sysv
call_h_sysv:
    mov rdi, 123            # # p is integer arg => uses RDI
     ↪  regardless of FP count
```

```
sub rsp, 8              # # reserve stack slot for i (9th FP arg)
# # store i as IEEE-754 double bits (illustrative; actual
↪  load/store may vary)
# # [rsp] = i
call h
add rsp, 8
ret
```

Discipline Summary

- Integer/pointer args: `RDI, RSI, RDX, RCX, R8, R9` (then stack).
- Floating-point args: `XMM0--XMM7` (then stack).
- Mixed lists use two independent sequences; do not “shift” XMM registers because of integer arguments or vice versa.
- Correct argument passing is a binary contract: the callee reads exactly what the caller places, and any mismatch is not recoverable.

Chapter 5

Stack Frame Layout

5.1 Stack Growth and Organization

In the System V AMD64 ABI, the stack is a contiguous region of memory used for call/return control flow, spilled registers, local storage, and passing arguments that do not fit in registers. Core properties:

- The stack **grows downward**: pushing data *decrements* `RSP`.
- `RSP` always points to the current top of the stack.
- A `call` instruction pushes the return address (8 bytes) onto the stack, decrementing `RSP` by 8.
- A `ret` instruction pops the return address into `RIP`, incrementing `RSP` by 8.

Example: How `call` and `ret` use the stack

```
.intel_syntax noprefix
```

```
.global demo_call_ret
demo_call_ret:
    # # Before call: RSP -> top
    call target         # # pushes return address: RSP -= 8; [RSP] =
     ↪  return RIP
    ret                 # # pops return address: RIP = [RSP]; RSP +=
     ↪  8

target:
    ret
```

5.2 Stack Frame Components

A **stack frame** is the portion of the stack owned by a function invocation. Not every function must create a visible frame, but any function that needs local storage, spills, or preservation will allocate one.

Typical frame components (from higher addresses to lower addresses):

- **Incoming stack arguments** (for arguments beyond those passed in registers)
- **Return address** (pushed by `call`)
- **Saved frame pointer** (optional: `push rbp`)
- **Saved callee-saved registers** (e.g., `RBX`, `R12--R15` if used)
- **Local variables / temporaries** (including compiler-created spill slots)
- **Alignment padding** (to satisfy ABI alignment rules)

Two practical facts:

- The exact layout varies by compiler, optimization, and whether a frame pointer is used.
- ABI **invariants** do not depend on a specific compiler layout: alignment and preservation rules must still hold.

Example: Frame with saved registers and locals (illustrative layout)

```
.intel_syntax noprefix
# High addresses
#    [arg7] [arg8] ...       # incoming stack args (if any)
#    [ret addr]              # pushed by call
#    [saved RBP]             # optional
#    [saved RBX]             # if used
#    [local/spill area]      # locals, spills, padding
# Low addresses
```

5.3 Function Prologue and Epilogue

A function prologue/epilogue establishes and tears down its frame and enforces ABI rules:

5.3.1 Common prologue pattern (with frame pointer)

- Save old `RBP` and set it to the current stack pointer
- Save any callee-saved registers the function will use
- Allocate stack space for locals/spills
- Maintain alignment rules for any further calls

```
.intel_syntax noprefix
```

```
.global func_with_frame
func_with_frame:
    push rbp
    mov  rbp, rsp

    push rbx            # # preserve callee-saved RBX if used
    sub  rsp, 32        # # locals/spills/padding (example size)

    # # ... function body ...
    add  rsp, 32
    pop  rbx
    pop  rbp
    ret
```

5.3.2 Leaf function pattern (no frame pointer)

A leaf function makes no calls. It can often avoid stack allocation completely if it uses only volatile registers and no local storage.

```
.intel_syntax noprefix
.global leaf_add
# long leaf_add(long a,long b)  # a=RDI, b=RSI, ret=RAX
leaf_add:
    lea rax, [rdi + rsi]
    ret
```

5.3.3 Non-leaf functions must respect call-site alignment

If a function makes calls, it must ensure `RSP` is in the correct state before each `call`. Even if the function uses no locals, a single `push` can break alignment and must be compensated.

```
.intel_syntax noprefix
.extern callee

.global nonleaf_alignment_example
nonleaf_alignment_example:
    push rbx            # # breaks alignment by 8
    sub  rsp, 8         # # fix alignment before call
    call callee
    add  rsp, 8
    pop  rbx
    ret
```

5.4 Red Zone Concept and Constraints

The System V AMD64 ABI defines a **red zone**: a 128-byte region *below* the current stack pointer (`RSP`) that is guaranteed not to be clobbered by signal/trap handlers in user-space on compliant systems.

- The red zone is located at addresses `[RSP - 128, RSP - 1]`.
- Leaf functions may use it for temporary storage **without adjusting `RSP`**.

This is a major difference from Windows x64, which has **no red zone**. Portable assembly must not rely on the red zone if it targets Windows.

5.4.1 Correct usage constraints

You may use the red zone only when all of the following are true:

- You are in System V AMD64 user-space (not Windows x64).

- You are in code where asynchronous events (signals/interrupt-like events) obey the red-zone guarantee.
- You do not modify `RSP` in a way that invalidates your offsets.
- You understand that **any function call** may overwrite your red-zone temporaries because callees may use their own stack frames and can adjust `RSP`.

Therefore, red-zone use is most natural in **leaf functions**.

Example: Leaf function using the red zone (no stack allocation)

```
.intel_syntax noprefix
.global redzone_leaf_demo
# long redzone_leaf_demo(long x)  # x=RDI
redzone_leaf_demo:
    # # Use 8 bytes in the red zone for a temporary.
    mov qword ptr [rsp - 8], rdi      # # store x
    add qword ptr [rsp - 8], 5        # # temp += 5
    mov rax, qword ptr [rsp - 8]      # # return temp
    ret
```

Incorrect usage: calling another function after storing in red zone

Once you call another function, the red-zone storage is no longer safe as “your” temporary area, because the callee may move `RSP` and overwrite memory below its own `RSP`.

```
.intel_syntax noprefix
.extern callee

.global redzone_call_bug
```

```
redzone_call_bug:
    mov qword ptr [rsp - 8], 123     # # store temp in red zone
    call callee                      # # callee may overwrite below
     ↪  its RSP
    mov rax, qword ptr [rsp - 8]     # # BUG: value may be destroyed
    ret
```

Discipline Summary

- The stack grows downward; `call` pushes the return address; `ret` pops it.
- A stack frame may include saved registers, locals/spills, and padding; exact layout varies, but ABI invariants must hold.
- Prologue/epilogue sequences implement preservation and stack management; non-leaf functions must maintain call-site alignment.
- The System V red zone provides 128 bytes below `RSP` usable mainly by leaf functions; it must not be relied upon for Windows targets and must not be treated as safe across calls.

Chapter 6

Stack Alignment Rules

6.1 16-Byte Alignment Requirement

The System V AMD64 ABI requires **16-byte stack alignment at call boundaries**. In practical terms:

- Immediately **before executing a `call`** instruction, `RSP` must be aligned to 16 bytes.
- The `call` instruction pushes an 8-byte return address, so **upon entry to the callee**, `RSP` is typically misaligned by 8 relative to 16 (i.e., `RSP % 16 == 8`).

This rule exists so that the callee can realign as needed and so that compilers can safely generate aligned stack spills, local allocations, and vector operations under stable assumptions. A concise mental model:

- **Call site:** `RSP % 16 == 0` (required)
- **Callee entry:** `RSP % 16 == 8` (because return address is pushed)

Example: Checking alignment by masking

```
.intel_syntax noprefix
.global rsp_mod16
rsp_mod16:
    mov rax, rsp
    and rax, 15          # # RAX = RSP % 16
    ret
```

6.2 Alignment at Function Call Boundaries

The alignment rule is a **caller responsibility**: the caller must ensure correct alignment *before* every call it makes.
This interacts with pushes, local allocations, and stack argument setup:

- Each `push` subtracts 8 bytes from `RSP` and toggles alignment between `0` and `8` mod 16.
- Any manual `sub rsp, N` must choose `N` such that alignment is correct at call sites.
- When placing stack arguments, the total stack adjustment must still preserve call-site alignment.

Example: A single push breaks call-site alignment

```
.intel_syntax noprefix
.extern callee

.global misaligned_call_due_to_push
misaligned_call_due_to_push:
    push rbx             # # RSP -= 8 => alignment toggles
    call callee          # # BUG: call-site alignment may be wrong
    pop  rbx
    ret
```

Fix: compensate with an extra 8-byte adjustment

```
.intel_syntax noprefix
.extern callee

.global aligned_call_with_compensation
aligned_call_with_compensation:
    push rbx
    sub  rsp, 8          # # restore 16-byte alignment before call
    call callee
    add  rsp, 8
    pop  rbx
    ret
```

Example: Non-leaf function allocating locals must still align before calls

```
.intel_syntax noprefix
.extern callee

.global locals_and_call
locals_and_call:
    push rbp
    mov  rbp, rsp

    # # Allocate 24 bytes of locals, but preserve call-site
     ↪  alignment.
    # # After 'push rbp', alignment flips; choose allocation with
     ↪  padding.
    sub  rsp, 32          # # 24 locals + 8 padding (example)

    call callee           # # aligned call site

    add  rsp, 32
    pop  rbp
    ret
```

Example: Passing stack arguments while preserving alignment

```
.intel_syntax noprefix
.extern f8

# long f8(long a,long b,long c,long d,long e,long f,long g,long h);

.global call_f8_aligned
call_f8_aligned:
    mov rdi, 1
    mov rsi, 2
    mov rdx, 3
    mov rcx, 4
    mov r8,  5
    mov r9,  6

    # # Need 16 bytes for g and h. This preserves 16-byte alignment
     ↪  if RSP was aligned here.
    sub rsp, 16
    mov qword ptr [rsp + 8], 8
    mov qword ptr [rsp + 0], 7

    call f8

    add rsp, 16
    ret
```

6.3 Effects of Misalignment on Execution

Misalignment is not merely “slower code”; it can produce severe failures depending on code generation, instruction selection, and library behavior.

6.3.1 Correctness failures

Misalignment may cause faults or undefined results when code uses instructions that require aligned memory operands or when library routines assume ABI-aligned stack frames. Common failure patterns:

- Crashes inside optimized library code (often appears unrelated to the caller).
- Intermittent failures that differ between debug/release builds.
- Failures that depend on CPU features (SSE/AVX usage) or compiler version.

Example: A callee using aligned stack spills (conceptual risk)

Even if your code never uses SIMD explicitly, the compiler may spill vector registers or local vectors to the stack with alignment assumptions. If the caller violates alignment, the callee may generate aligned moves that fault or misbehave.

```
.intel_syntax noprefix
# Conceptual illustration:
# If a compiler assumes alignment, it may emit aligned loads/stores
↪  (e.g., for 16-byte objects).
# If the stack is misaligned, those aligned accesses can become
↪  invalid at runtime.
```

6.3.2 Performance degradation

Even when misalignment does not crash, it can still degrade performance:

- Some unaligned loads/stores are slower or require more micro-operations.
- Misalignment can force the compiler to generate more conservative code.
- Misaligned stack frames can increase register spill traffic and reduce vector efficiency.

Example: A hidden performance bug

A function that accidentally misaligns the stack may cause all callees in a hot loop to pay extra penalties due to conservative spills or unaligned memory operations.

```
.intel_syntax noprefix
.extern hot_callee

.global hidden_alignment_perf_bug
hidden_alignment_perf_bug:
    push rbx             # # breaks alignment
    # # missing alignment fix here
    call hot_callee      # # may still run, but with avoidable
     ↪  penalties
    pop   rbx
    ret
```

6.3.3 Debugging symptoms

Alignment bugs often manifest as:

- Crashes in unrelated functions (because the failure happens after an ABI violation).
- Stack traces that look corrupted (because an ABI violation can cascade into wrong unwinding).
- “Only fails with optimization” or “only fails on some machines” due to different instruction choices.

Discipline Summary

- System V AMD64 requires `RSP` to be **16-byte aligned before every `call`**.
- The caller is responsible for alignment; pushes and local allocations must be balanced with padding.
- Misalignment can cause crashes, corrupt execution, destroy stack traces, or silently degrade performance.
- Always audit your assembly and FFI boundaries: treat alignment as a hard correctness rule, not an optimization.

Chapter 7

Caller-Saved vs Callee-Saved Registers

7.1 Preservation Responsibilities

In the System V AMD64 ABI, registers are divided by **preservation responsibility**:

- **Caller-saved (volatile):** the caller must assume the callee may overwrite these registers. If the caller needs their values after a call, it must save them before the call and restore them after.
- **Callee-saved (non-volatile):** the callee must preserve these registers. If the callee uses them, it must save their incoming values and restore them before returning.

System V AMD64 GPR classification

Callee-saved (must be preserved by callee):

```
RBX, RBP, R12, R13, R14, R15
```

Caller-saved (may be clobbered by callee):

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

Special discipline rules:

- **RSP** must always be restored to its incoming value before `ret`.
- **Vector registers (XMM/YMM)** are caller-saved under System V AMD64 (caller must save if needed across calls).

Why this rule exists

This split balances performance and composability:

- Most calls can be fast because callees are not forced to save everything.
- Callers only save what they actually need to keep.
- Callee-saved registers provide stable long-lived storage across calls (useful for local state in non-leaf functions).

7.2 Register Save and Restore Mechanics

Save/restore mechanics must preserve values exactly, keep stack discipline correct, and maintain call-site alignment when making further calls.

7.2.1 Callee saving callee-saved registers

If a callee uses a callee-saved register, it must save it early (typically in the prologue) and restore it late (typically in the epilogue).

Example: Callee uses RBX and must preserve it

```
.intel_syntax noprefix
```

```
.global callee_uses_rbx
callee_uses_rbx:
    push rbx              # # save callee-saved RBX
    # # ... RBX is now safe to use ...
    mov  rbx, 123
    # # ... work ...
    pop  rbx              # # restore RBX before return
    ret
```

If the callee uses multiple callee-saved registers, it must preserve all it touches:

```
.intel_syntax noprefix
.global callee_uses_r12_r13
callee_uses_r12_r13:
    push r12
    push r13
    # # ... use r12/r13 ...
    pop  r13
    pop  r12
    ret
```

7.2.2 Caller saving caller-saved registers

If the caller needs a volatile register after a call, it must save it itself. The simplest mechanism is to spill to the stack (or move into a callee-saved register that the caller owns across its own calls).

Example: Caller preserves R10 across a call

```
.intel_syntax noprefix
```

```
.extern callee

.global caller_needs_r10
caller_needs_r10:
    mov  r10, 0xABCDEF

    push r10              # # save volatile register (caller
     ↪  responsibility)
    call callee
    pop  r10              # # restore after call

    mov  rax, r10
    ret
```

7.2.3 Maintaining stack alignment while saving registers

Because each `push` changes `RSP` by 8 bytes, saving an *odd* number of registers can break call-site alignment. If the function makes calls, it must compensate.

Example: Callee saves RBX then calls another function (alignment disciplined)

```
.intel_syntax noprefix
.extern other

.global callee_save_then_call
callee_save_then_call:
    push rbx              # # save callee-saved
    sub  rsp, 8           # # maintain 16-byte alignment before call
    call other
    add  rsp, 8
    pop  rbx
    ret
```

7.2.4 Saving vector registers in System V AMD64

All XMM/YMM registers are caller-saved. If a caller needs an XMM register value preserved, it must save/restore it.

```
.intel_syntax noprefix
.extern callee

.global caller_preserves_xmm1
caller_preserves_xmm1:
    sub  rsp, 16
    movdqu [rsp], xmm1      # # save volatile XMM1
    call callee
    movdqu xmm1, [rsp]      # # restore
    add  rsp, 16
    ret
```

7.3 Practical Calling Scenarios

7.3.1 Scenario 1: Non-leaf function needs stable locals across calls

Use callee-saved registers for long-lived local state within the function; save them once in the prologue and restore once in the epilogue.

```
.intel_syntax noprefix
```

```
.extern step1
.extern step2

.global pipeline_example
# long pipeline_example(long x)
pipeline_example:
    push rbx
    sub  rsp, 8             # # align before calls

    mov  rbx, rdi           # # keep x in RBX across multiple calls

    mov  rdi, rbx
    call step1

    mov  rdi, rbx
    call step2

    add  rsp, 8
    pop  rbx
    ret
```

Here, `RBX` is used as a stable local variable because it is callee-saved and thus remains valid across calls inside the same function (after we preserved it in the prologue).

7.3.2 Scenario 2: Caller wants to keep an argument register value after calling

Argument registers are caller-saved under SysV. If the caller wants to reuse `RDI` or `RSI` after calling another function, it must save them.

```
.intel_syntax noprefix
.extern callee

.global reuse_rdi_after_call
reuse_rdi_after_call:
    # # RDI holds an important pointer we need after the call
    push rdi
    call callee
    pop  rdi                # # restore our pointer
    # # safe to use RDI again
    mov  rax, rdi
    ret
```

7.3.3 Scenario 3: Bug pattern — callee clobbers callee-saved register

If a callee clobbers `RBX` without preserving it, it violates the ABI and breaks any caller that relies on `RBX` surviving across the call.

```
.intel_syntax noprefix
.global buggy_callee_clobber_rbx
buggy_callee_clobber_rbx:
    mov rbx, 999            # # BUG: RBX is callee-saved; must be
     ↪  preserved
    ret
```

Correct version:

```
.intel_syntax noprefix
.global fixed_callee_preserve_rbx
fixed_callee_preserve_rbx:
    push rbx
    mov   rbx, 999
    pop   rbx
    ret
```

7.3.4 Scenario 4: Leaf functions and the red zone

Leaf functions that make no calls can often avoid stack frame allocation, and may optionally use the red zone for temporary storage. This does not change preservation rules: caller-saved registers are still volatile across any call (but leaf functions do not call).

```
.intel_syntax noprefix
.global leaf_uses_volatiles
# long leaf_uses_volatiles(long a, long b)
leaf_uses_volatiles:
    # # Safe: leaf uses volatile registers and makes no calls
    lea rax, [rdi + rsi]
    ret
```

Discipline Summary

- Callee must preserve: `RBX, RBP, R12--R15`.
- Caller must preserve if needed: `RAX, RCX, RDX, RSI, RDI, R8--R11` and all `XMM/YMM` registers.
- Save/restore must maintain stack correctness and call-site 16-byte alignment.
- Use callee-saved registers for long-lived locals in non-leaf functions; save caller-saved registers only when the caller truly needs them after a call.

Chapter 8

Function Call and Return Mechanics

8.1 `call` and `ret` Instruction Behavior

In x86-64, function calls and returns are implemented using the `call` and `ret` instructions. These instructions manipulate the stack and instruction pointer directly and form the foundation of all ABI-level control flow.

8.1.1 Behavior of `call`

The `call` instruction performs two atomic actions:

- Pushes the **return address** (the address of the instruction following `call`) onto the stack.
- Transfers control to the call target by loading its address into `RIP`.

Formally:

- `RSP := RSP - 8`

- `[RSP] := RIP_next`
- `RIP := target`

8.1.2 Behavior of `ret`

The `ret` instruction reverses this process:

- Pops the return address from the stack into `RIP`.
- Increments `RSP` by 8.

Formally:

- `RIP := [RSP]`
- `RSP := RSP + 8`

Example: Minimal call/return pair

```
.intel_syntax noprefix
.global caller
caller:
    call callee            # # pushes return address, jumps to callee
    ret

callee:
    ret                    # # pops return address, returns to caller
```

8.2 Return Address Handling

The return address is an **implicit stack argument** managed entirely by the CPU. It is not passed in a register and must be treated as read-only control-flow data.
Key rules:

- The return address always resides at the top of the stack on function entry.
- It must remain intact until `ret` executes.
- Any stack manipulation must preserve its position relative to `RSP`.

Example: Stack layout at function entry

```
.intel_syntax noprefix
# On entry to callee:
#   RSP -> [ return address ]
```

8.2.1 Saving and restoring around the return address

If a function needs stack space or must save registers, it must do so *below* the return address by adjusting `RSP`.

Correct: allocating locals below the return address

```
.intel_syntax noprefix
.global func_with_locals
func_with_locals:
    sub  rsp, 16           # # allocate local space below return
     ↪  address
    # # ... use locals ...
    add  rsp, 16
    ret
```

Incorrect: overwriting the return address

```
.intel_syntax noprefix
.global overwrite_retaddr_bug
overwrite_retaddr_bug:
    mov qword ptr [rsp], 0 # # BUG: overwrites return address
    ret                    # # undefined control flow
```

8.3 Stack State Before and After Calls

Understanding stack state transitions is critical for ABI correctness.

8.3.1 Stack state before `call`

Before executing `call`, the caller must ensure:

- All arguments are placed in the correct registers and/or stack slots.
- `RSP` is **16-byte aligned**.
- Any required caller-saved registers are preserved.

Example: Correct pre-call setup

```
.intel_syntax noprefix
.extern callee

.global caller_setup
```

```
caller_setup:
    # # RSP is 16-byte aligned here
    mov rdi, 10              # # argument 1
    call callee
    ret
```

8.3.2 Stack state immediately after `call`

Once `call` executes:

- `RSP` is decremented by 8.
- The return address is at `[RSP]`.
- `RSP % 16 == 8` on callee entry (assuming correct call-site alignment).

Example: Entry state inside callee

```
.intel_syntax noprefix
callee_entry:
    # # RSP points to return address
    # # RSP % 16 == 8
    ret
```

8.3.3 Stack state before `ret`

Before executing `ret`, the callee must ensure:

- All callee-saved registers have been restored.
- `RSP` has been restored to its incoming value.
- The return address remains at the top of the stack.

Example: Correct epilogue

```
.intel_syntax noprefix
.global func_epilogue
func_epilogue:
    # # ... body ...
    add  rsp, 16           # # deallocate locals
    ret
```

8.3.4 Stack state after `ret`

After `ret`:

- `RIP` resumes at the caller's next instruction.
- `RSP` has the same value it had immediately before the original `call`.

Example: Caller resumes execution

```
.intel_syntax noprefix
caller_resume:
    call callee
    # # execution resumes here after ret
    ret
```

8.4 Nested Calls and Stack Integrity

Nested calls form a stack of return addresses. Each `call` pushes a new return address; each `ret` pops exactly one.

Example: Nested calls

```
.intel_syntax noprefix
.global f1
.global f2
.global f3

f1:
    call f2
    ret

f2:
    call f3
    ret

f3:
    ret
```

The return sequence is strictly LIFO:

$$f3 \to f2 \to f1$$

Any imbalance in stack adjustments breaks this chain.

Discipline Summary

- `call` pushes the return address and transfers control.
- `ret` pops the return address and restores control.
- The return address must never be overwritten or skipped.

- Callers must align the stack before calls; callees must restore it before returns.
- Stack imbalance or return-address corruption results in immediate undefined control flow.

Chapter 9

Returning Values and Aggregates

9.1 Scalar Return Values

Under the System V AMD64 ABI, **scalar return values** are returned in registers. The rules are simple, strict, and heavily relied upon by compilers and debuggers.

- **Integer and pointer returns:** `RAX`
- **Boolean returns:** `AL` (observed via `RAX` as 0 or 1)
- **Floating-point returns:** `XMM0`

For integers smaller than 64 bits, the ABI relies on **type-directed extension**:

- Signed types are sign-extended to 64 bits.
- Unsigned types are zero-extended to 64 bits.

Example: Returning a 64-bit integer

```
.intel_syntax noprefix
.global ret_i64
ret_i64:
    mov rax, 0x1122334455667788
    ret
```

Example: Returning a boolean

```
.intel_syntax noprefix
.global is_nonzero
# long is_nonzero(long x)   # x in RDI
is_nonzero:
    xor eax, eax
    test rdi, rdi
    setne al                # # AL = 1 if x != 0
    ret
```

Example: Returning a double

```
.intel_syntax noprefix
.global ret_double
ret_double:
    # # return value must be in XMM0
    ret
```

9.2 Structure and Union Return Rules

Returning aggregates (structures or unions) is governed by **size and classification**. The ABI classifies aggregates into register-returnable or memory-returned categories.
Core rules:

- Aggregates of size **up to 16 bytes** may be returned in registers.
- Aggregates larger than 16 bytes are returned via memory using a hidden pointer.
- Register-returnable aggregates are decomposed into up to two eight-byte chunks.

Each eight-byte chunk is classified independently:

- INTEGER class → `RAX` / `RDX`
- SSE class → `XMM0` / `XMM1`

Example: Returning a small integer-only struct (8 bytes)

```
.intel_syntax noprefix
# struct S { long x; };

.global ret_struct_8
ret_struct_8:
    mov rax, 123           # # entire struct in RAX
    ret
```

Example: Returning a 16-byte integer struct

```
.intel_syntax noprefix
# struct S { long a; long b; };
```

```
.global ret_struct_16
ret_struct_16:
    mov rax, 1              # # first field
    mov rdx, 2              # # second field
    ret
```

Example: Returning a mixed FP/integer struct

Conceptual type:

```
.intel_syntax noprefix
# struct M { double x; long y; };
```

Register assignment:

- `x` in `XMM0`
- `y` in `RAX`

```
.intel_syntax noprefix
.global ret_mixed_struct
ret_mixed_struct:
    # # XMM0 holds double field
    mov rax, 42             # # integer field
    ret
```

9.3 Hidden Pointer Mechanism

When an aggregate cannot be returned in registers, the ABI uses a **hidden return pointer** supplied by the caller.
Rules:

- The caller allocates space for the return object.
- A pointer to this space is passed as an implicit first argument.
- The callee writes the result into the pointed memory.
- The function formally returns `void`.

In System V AMD64:

- The hidden pointer is passed in `RDI`.
- User-visible arguments are shifted to the next registers.

Example: Large struct returned via hidden pointer

Conceptual type:

```
.intel_syntax noprefix
# struct Big { long a; long b; long c; };
# Big make_big(void);
```

Calling convention:

- `RDI` points to caller-allocated return storage.

```
.intel_syntax noprefix
.global make_big
make_big:
    mov qword ptr [rdi + 0], 1
    mov qword ptr [rdi + 8], 2
    mov qword ptr [rdi + 16], 3
    ret
```

9.4 Large Object Return Handling

Large aggregates (greater than 16 bytes, or those failing register classification) are always returned via memory. This mechanism is fundamental to ABI stability and cross-language interoperability.
Key implications:

- The callee does not allocate return storage.
- Lifetime and alignment of the return object are the caller's responsibility.
- Debuggers and exception-unwinding logic rely on this convention.

Caller-side perspective (conceptual)

```
.intel_syntax noprefix
# Caller allocates space, passes pointer in RDI, then calls.
# Returned object is already written on return.
```

Common bug: forgetting the hidden pointer

If handwritten assembly omits the hidden pointer or misplaces arguments, the callee writes into an invalid address, often corrupting memory silently.

```
.intel_syntax noprefix
# BUG: calling make_big without setting RDI to valid storage
# leads to memory corruption.
```

Discipline Summary

- Scalars return in `RAX` (integers/pointers) or `XMM0` (floating-point).

- Aggregates up to 16 bytes may return in registers, split into 8-byte units.
- Larger or non-classifiable aggregates use a hidden return pointer in `RDI`.
- Correct handling of aggregate returns is mandatory for ABI correctness and interoperability.

Chapter 10

Variadic Functions (System V Basics)

10.1 Default Argument Promotions

In C/C++ variadic functions (functions declared with `...`), the compiler cannot infer the types of the unnamed arguments from the callee side alone. Therefore, the language applies **default argument promotions** at the call site, and the callee must retrieve arguments using an explicit type protocol (typically `va_list` and `va_arg`).
Core promotions (C language rules, inherited by C++ varargs):

- **Integer promotions:** `char`, `signed char`, `unsigned char`, `short`, `unsigned short` are promoted to `int` or `unsigned int`.
- **Floating promotions:** `float` is promoted to `double`.

Practical implications for ABI-level reasoning:

- A variadic callee must **never** attempt to read a `float` from the variadic pack; it will be present as a `double`.

- A variadic callee must **never** attempt to read a `char/short` from the variadic pack; it will be present as an `int`.

Example: Promotion consequences (conceptual)

```
.intel_syntax noprefix
# In a call like:
#   v("x", (char)1, (short)2, (float)3.0f);
# The variadic payload is actually:
#   int 1, int 2, double 3.0
```

10.2 Register Save Area

In the System V AMD64 ABI, variadic functions require a special mechanism so that a callee can reliably access arguments regardless of whether the caller placed them in registers or on the stack.
Key ABI idea:

- Named arguments follow normal register/stack passing.
- Variadic access (`va_arg`) needs a uniform memory view of register-passed arguments.
- Therefore, a variadic callee typically uses a **register save area** (also called the **register argument home area**) where incoming register arguments are spilled so they can be walked like memory.

Practical model used by compilers:

- The callee saves (copies) the incoming argument registers into a local memory block.
- The `va_list` structure maintains offsets/pointers indicating:

  - where the next integer-class variadic argument is (GPR path),
  - where the next SSE-class variadic argument is (XMM path),
  - where stack overflow arguments begin.

You do not need the concrete internal layout of `va_list` to write correct assembly at a boundary; you must, however, respect that:

- the callee may read register-passed arguments from its own save area,
- the callee may read overflow arguments from the caller-provided stack area.

Example: Why a save area is necessary

```
.intel_syntax noprefix
# Without saving register arguments to memory, va_arg would have to:
# - read "the next argument" sometimes from registers and sometimes
↪  from stack,
# - but registers are not an addressable stream,
# - and the callee may already have overwritten them.
# The save area turns register arguments into a stable memory stream.
```

10.3 Accessing Variadic Arguments

At the source level, variadic arguments are accessed using:

- `va_start`
- `va_arg`
- `va_end`

At the ABI level, **two independent streams exist**:

- **GPR stream** for integer/pointer class arguments
- **SSE stream** for floating-point arguments

A variadic callee selects the stream based on the type requested in `va_arg`.

Example: C-like access pattern (conceptual)

```
.intel_syntax noprefix
# Conceptual:
#   int    i = va_arg(ap, int);    # from GPR save area or stack
↪   overflow
#   double d = va_arg(ap, double); # from XMM save area or stack
↪   overflow
```

Assembly boundary discipline: preserve argument registers early

If you implement a variadic function in assembly (or mix inline assembly), you must assume:

- Argument registers used for the named parameters are live on entry.
- Additional register arguments for the variadic payload may also be live.
- If you overwrite these registers before saving them, you destroy the variadic payload.

```
.intel_syntax noprefix
.global vfunc_save_early_demo
vfunc_save_early_demo:
    # # Discipline: if you plan to parse variadic args, preserve the
    ↪  incoming regs early.
```

```
# # Example saves of a few registers; real implementations save
 ↪  all needed ABI argument regs.
push rbp
mov  rbp, rsp
sub  rsp, 128           # # local save area (illustrative size)

mov  [rbp - 8], rdi     # # save first integer arg register
mov  [rbp - 16], rsi
mov  [rbp - 24], rdx
mov  [rbp - 32], rcx
mov  [rbp - 40], r8
mov  [rbp - 48], r9

# # similarly, XMM0..XMM7 may need to be saved for FP variadic
 ↪  access
# # movdqu [rbp - 64], xmm0   # # etc.

# # ... parse arguments using a protocol ...
leave
ret
```

10.4 ABI Limitations and Pitfalls

Variadic functions are powerful but fragile. Many pitfalls are not “logic bugs”; they are ABI and type-protocol violations.

10.4.1 Pitfall 1: Reading the wrong promoted type

If a caller passed `float` in a variadic position, the callee must read a `double`. Reading a `float` is invalid and will consume the wrong number of bytes/slots, desynchronizing subsequent reads.

```
.intel_syntax noprefix
# Bug pattern (conceptual):
#   float f = va_arg(ap, float);   # wrong: float was promoted to
↪  double
# Correct:
#   double d = va_arg(ap, double);
```

10.4.2 Pitfall 2: Assuming a single linear stream

Because SysV maintains separate GPR and SSE tracks, reading types in a different order than the caller used is catastrophic. The callee must follow the exact protocol of types.

```
.intel_syntax noprefix
# If the caller did:
#   v("%d %f", 7, 3.5);
# The callee must read:
#   int then double
# Not:
#   double then int
```

10.4.3 Pitfall 3: Implementing variadic functions in assembly without saving XMM regs

Even if named parameters are integer-only, a variadic payload may include floating-point arguments. If the callee overwrites XMM registers before capturing them into a save area, those variadic FP values are lost.

```
.intel_syntax noprefix
.global clobber_xmm_bug_variadic
clobber_xmm_bug_variadic:
    # # BUG: overwrites XMM0 before saving it
    xorps xmm0, xmm0
    ret
```

10.4.4 Pitfall 4: Cross-ABI porting assumptions

System V variadic mechanics differ from Windows x64 (register count, home space, and `va_list` layout rules differ). Variadic functions are one of the quickest ways to break code when porting.

```
.intel_syntax noprefix
# Discipline: treat SysV varargs and Win64 varargs as different ABIs.
# Never assume a va_list layout or register-save strategy transfers
↪  across platforms.
```

10.4.5 Pitfall 5: Passing aggregates through ...

Passing structs/unions through `...` is especially risky because the callee cannot infer the layout or classification unless the protocol is explicitly defined. Robust interfaces avoid aggregates in variadic positions and instead pass pointers or use typed wrappers.

```
.intel_syntax noprefix
# Safer design pattern:
# - pass pointer to struct, or
# - use a typed API rather than varargs for non-trivial data.
```

Discipline Summary

- Default promotions: small integers $\rightarrow$ `int`, `float` $\rightarrow$ `double`.
- SysV variadic access depends on a register save area to provide a stable memory view of register-passed args.
- Variadic access is type-driven and uses separate tracks for GPR and SSE arguments; the callee must follow the exact type protocol.
- Major pitfalls: wrong promoted types, wrong read order, clobbering XMM regs early, and cross-ABI assumptions.

Chapter 11

Common ABI Violations and Debugging Symptoms

11.1 Stack Corruption Indicators

Stack corruption means the call/return chain or stack-owned data has been modified incorrectly. Under System V AMD64, the most common root causes are:

- **Unbalanced stack adjustments:** `sub rsp, N` not matched by `add rsp, N`, or mismatched `push`/`pop`.
- **Overwriting the return address:** writing to `[rsp]` (or `[rbp+8]` when using a frame pointer).
- **Writing beyond allocated locals:** out-of-bounds stack writes (e.g., using `[rsp-8]` without red-zone guarantees, or indexing into locals incorrectly).
- **Wrong stack-argument offsets:** reading stack arguments using incorrect offsets due to missing frame pointer assumptions or wrong allocation size.

Typical symptoms:

- Crash on `ret` (jumping to a garbage return address).
- Backtrace stops early, shows nonsense addresses, or unwinds incorrectly.
- "Works in debug, crashes in release" because the compiler's stack frame differs.
- Failure appears inside unrelated library code because the ABI violation occurred earlier.

Example: Unbalanced stack adjustment breaks return

```
.intel_syntax noprefix
.global unbalanced_stack_bug
unbalanced_stack_bug:
    sub rsp, 16           # # allocate locals
    # # ... work ...
    # BUG: missing add rsp,16
    ret                   # # ret pops wrong address (stack shifted)
```

Correct version:

```
.intel_syntax noprefix
.global balanced_stack_ok
balanced_stack_ok:
    sub rsp, 16
    # # ... work ...
    add rsp, 16
    ret
```

Example: Overwriting the return address

```
.intel_syntax noprefix
.global overwrite_retaddr_bug
overwrite_retaddr_bug:
    mov qword ptr [rsp], 0x0    # # BUG: overwrites return address
    ret                         # # undefined control flow
```

Example: Wrong local offset corrupts saved state

```
.intel_syntax noprefix
.global wrong_offset_bug
wrong_offset_bug:
    push rbp
    mov  rbp, rsp
    sub  rsp, 16

    # Suppose intended local is at [rbp-8], but a bug writes [rbp+8]
     ↪  instead.
    # [rbp+8] is the return address in the canonical frame layout.
    mov qword ptr [rbp + 8], 0x4141414141414141  # # BUG

    leave
    ret
```

11.2 Register Clobbering Errors

Register clobbering errors occur when caller/callee preservation responsibilities are violated. System V AMD64 rules (GPRs):

- **Callee-saved:** `RBX, RBP, R12--R15` (callee must preserve)
- **Caller-saved:** `RAX, RCX, RDX, RSI, RDI, R8--R11` (caller must preserve if needed)

Vector registers:

- **XMM/YMM are caller-saved** under System V AMD64.

Typical symptoms:

- Values "randomly" change after calls.
- Bugs appear only with inlining disabled or with different optimization levels.
- Errors reproduce only when a particular call path executes (because clobber happens in one callee).

Example: Callee clobbers RBX (ABI violation)

```
.intel_syntax noprefix
.global clobber_rbx_bug
clobber_rbx_bug:
    mov rbx, 999          # # BUG: RBX is callee-saved on SysV
    ret
```

Correct version:

```
.intel_syntax noprefix
.global preserve_rbx_ok
preserve_rbx_ok:
    push rbx
    mov  rbx, 999
    pop  rbx
    ret
```

Example: Caller assumes RDI survives a call (wrong on SysV)

```
.intel_syntax noprefix
.extern callee

.global caller_wrong_rdi_assumption
caller_wrong_rdi_assumption:
    mov rdi, 77
    call callee            # # callee may clobber RDI
    mov rax, rdi           # # BUG: assumes RDI still 77
    ret
```

Correct caller strategy (save volatile register if needed):

```
.intel_syntax noprefix
.extern callee

.global caller_saves_rdi_ok
caller_saves_rdi_ok:
    mov  rdi, 77
    push rdi
    call callee
    pop  rdi
    mov  rax, rdi
    ret
```

Example: Caller forgets XMM registers are volatile

```
.intel_syntax noprefix
.extern callee
```

```
.global caller_forgets_xmm0_bug
caller_forgets_xmm0_bug:
    # # XMM0 holds an important value here
    call callee             # # callee may clobber XMM0 (SysV:
     ↪  caller-saved)
    ret
```

Correct preservation:

```
.intel_syntax noprefix
.extern callee

.global caller_saves_xmm0_ok
caller_saves_xmm0_ok:
    sub  rsp, 16
    movdqu [rsp], xmm0
    call callee
    movdqu xmm0, [rsp]
    add  rsp, 16
    ret
```

11.3 Alignment-Related Crashes

System V AMD64 requires **16-byte alignment at call sites**. Violations often produce failures that look unrelated to the violating code.

Common causes:

- A `push` or odd number of pushes before a call without compensation.
- Manual local allocation sizes that do not preserve alignment.

- Incorrect stack argument reservation sizes.

Typical symptoms:

- Crash inside optimized code that uses vector spills or assumes aligned stack.
- Crash appears only on some CPUs or only with certain compiler flags.
- Unwinding/backtrace breaks when combined with other ABI violations.

Example: Misaligned call site due to one push

```
.intel_syntax noprefix
.extern callee

.global misaligned_call_bug
misaligned_call_bug:
    push rbx              # # toggles alignment
    call callee           # # BUG: call-site may violate 16-byte
     ↪  alignment
    pop  rbx
    ret
```

Correct version (compensate with 8 bytes):

```
.intel_syntax noprefix
.extern callee

.global aligned_call_ok
aligned_call_ok:
    push rbx
    sub  rsp, 8           # # restore alignment before call
    call callee
    add  rsp, 8
    pop  rbx
    ret
```

Example: Misalignment caused by wrong local allocation size

```
.intel_syntax noprefix
.extern callee

.global alloc_24_then_call_bug
alloc_24_then_call_bug:
    push rbp
    mov  rbp, rsp
    sub  rsp, 24          # # BUG: may break call-site alignment
     ↪  depending on entry state
    call callee
    leave
    ret
```

Corrected pattern: choose allocation with padding to satisfy alignment.

```
.intel_syntax noprefix
.extern callee

.global alloc_32_then_call_ok
alloc_32_then_call_ok:
    push rbp
    mov  rbp, rsp
```

```
sub  rsp, 32          # # locals + padding (keeps call-site
 ↪  alignment)
call callee
leave
ret
```

Debugging Checklist

When debugging an ABI symptom, validate these invariants first:

- **Stack balance:** every `sub rsp,N` has a matching `add rsp,N`; every `push` has a matching `pop`.
- **Return address integrity:** never write to `[rsp]` (or `[rbp+8]` with frame pointer).
- **Register preservation:** callee preserves `RBX, RBP, R12--R15`; caller saves volatile registers it needs.
- **Call-site alignment:** `RSP` is 16-byte aligned before each `call`.
- **XMM/YMM volatility:** caller saves vector registers if needed across calls.

Chapter 12

Interoperability and Language Boundaries

12.1 C and C++ ABI Interaction

In System V AMD64 user space, the **calling convention** for ordinary C functions is stable and well-defined at the ABI level: argument registers, return registers, stack discipline, and register preservation rules follow the System V AMD64 ABI.
C++ introduces additional binary-level mechanisms beyond the base calling convention, including:

- **Name mangling:** C++ encodes function names with type information.
- **Overloading:** multiple functions with the same source name require mangled symbols.
- **Member functions:** an implicit `this` pointer argument is passed like a normal pointer argument.
- **Constructors/destructors:** may have multiple entry points and special calling patterns.
- **Exceptions and unwinding:** require runtime support and strict stack frame/unwind metadata correctness.

Therefore, the safest interoperability boundary is **C linkage**.

Rule: Use C linkage for external boundaries

- Export boundary functions using C linkage so symbol names remain stable.
- Keep the boundary signature "plain": integers, pointers, POD-like data, explicit sizes.
- Avoid C++ features at the boundary (overloads, templates, references, exceptions).

Example: Stable boundary symbol (conceptual)

```
.intel_syntax noprefix
# In C++ source, the boundary is declared with C linkage:
# extern "C" long api_add(long a, long b);
#
# The SysV ABI then ensures:
#   a in RDI, b in RSI, return in RAX
```

Assembly implementation of a C-linkage boundary

```
.intel_syntax noprefix
.global api_add
# long api_add(long a, long b)  # SysV: a=RDI, b=RSI
api_add:
    lea rax, [rdi + rsi]
    ret
```

C++ member functions: implicit `this`

A non-static member function receives a hidden first argument: the `this` pointer. Under SysV, it is passed like the first pointer argument (`RDI`).

```
.intel_syntax noprefix
# Conceptual:
# struct Obj { long x; long get() const; };
# get(this) is called with:
#   this in RDI
```

```
.intel_syntax noprefix
.global obj_get_x
# long obj_get_x(Obj* this)  # explicit form used at ABI boundary
obj_get_x:
    mov rax, qword ptr [rdi + 0]   # # load this->x
    ret
```

12.2 Inline Assembly Constraints

Inline assembly is not “just assembly”; it is a contract with the compiler. The compiler is still responsible for register allocation, scheduling, stack layout, and ABI compliance across the whole function.
Key constraints:

- You must declare all **clobbered registers** correctly (or the compiler may assume they are unchanged).
- You must respect **stack alignment** and **callee-saved preservation** rules if your inline assembly performs calls or changes preserved state.

- You must treat inline assembly as a **black box** to the optimizer unless constraints are precise; incorrect constraints lead to miscompilation.

Example: The core danger — undeclared clobber

If inline assembly modifies a register that the compiler thinks is still live, the program can break without any visible assembly error.

```
.intel_syntax noprefix
# Conceptual bug:
# - compiler keeps a live value in RBX across the asm block
# - asm clobbers RBX
# - compiler later uses the corrupted value
```

ABI discipline for inline asm that calls another function

If your inline assembly emits a `call`, it must behave like any other caller:

- Provide correct argument registers.
- Preserve any caller-saved registers needed after the call.
- Maintain 16-byte stack alignment at the call site.

```
.intel_syntax noprefix
.extern callee

.global inline_style_call_example
inline_style_call_example:
    # # Treat this as if it were inline asm inside a compiler-managed
     ↪  function:
    # # 1) align stack before call
    # # 2) assume volatile regs may be clobbered
    sub rsp, 8              # # alignment compensation (illustrative)
    call callee
    add rsp, 8
    ret
```

12.3 Cross-Language Calling Safety

Cross-language calls are safe only if **both sides agree** on:

- calling convention (SysV AMD64),
- type sizes and alignment,
- structure layout rules,
- name binding (symbol names),
- ownership and lifetime rules for pointers and buffers,
- exception/error propagation rules.

12.3.1 Safe design patterns

Pattern 1: C ABI boundary wrapper

- Use a C ABI boundary even if the implementation is C++.
- Export only flat functions with explicit pointers and lengths.
- Return errors via integer codes, not exceptions.

Example: “buffer + length” boundary

```
.intel_syntax noprefix
# Conceptual C ABI:
#   long api_process(const unsigned char* buf, long len);
# SysV: buf in RDI, len in RSI
```

```
.intel_syntax noprefix
.global api_process
api_process:
    # # validate inputs without exceptions
    test rdi, rdi
    je   .bad
    test rsi, rsi
    jle  .bad
    # # ... process ...
    xor eax, eax          # # success => 0
    ret
.bad:
    mov eax, -1           # # error => -1
    ret
```

Pattern 2: Opaque handles

- Do not expose C++ object layout across the boundary.
- Pass an opaque pointer/handle and provide constructor/destructor functions.

Example: Handle-based ABI (conceptual)

```
.intel_syntax noprefix
```

```
# Conceptual C ABI:
#   void* obj_create();
#   void  obj_destroy(void* h);
#   long  obj_do(void* h, long x);
#
# The foreign side never depends on the C++ class layout.
```

12.3.2 Pitfalls and failure modes

Pitfall 1: C++ exceptions across non-C++ boundaries

- Throwing across an FFI boundary is unsafe unless both sides share the same unwinding model and runtime.
- A safe boundary converts exceptions to error codes.

Pitfall 2: Returning aggregates without matching ABI classification

- If two languages disagree on struct layout or return classification, the caller and callee will use different registers/memory.
- Prefer returning simple scalars and writing complex results into caller-provided buffers.

Pitfall 3: Variadic interfaces across languages

- Variadic conventions depend on language-level promotions and `va_list` layout.
- Do not use varargs as a cross-language boundary; use explicit typed APIs instead.

Example: Safe alternative to returning a large object

```
.intel_syntax noprefix
# Instead of:
```

```
#   Big make_big();
# Prefer:
#   long make_big(Big* out);
# where 'out' is a caller-allocated buffer and the return value is a
↪  status code.
```

Boundary Checklist

- Use C linkage for exported symbols and keep boundary signatures simple.
- Avoid C++-only features (overloads, templates, exceptions, references) across the boundary.
- Treat inline assembly as a compiler contract: declare clobbers and preserve ABI invariants.
- Prefer buffer+length and handle-based designs over exposing layouts or using varargs.
- Never assume portability of ABI details across operating systems or toolchains.

Chapter 13

ABI Discipline Checklist

13.1 Mandatory Rules Recap

This chapter is a compact, enforcement-oriented checklist for System V AMD64 correctness. Treat every item as mandatory when writing assembly, inline assembly, JIT code, or cross-language boundaries.

Argument passing (register-based)

- Integer/pointer args (first 6): `RDI, RSI, RDX, RCX, R8, R9`
- Floating-point args (first 8): `XMM0--XMM7`
- Mixed lists: two independent streams (INTEGER and SSE); do not shift one stream because of the other.
- Remaining args after registers: passed on the stack with correct layout and alignment.

Return values

- Integer/pointer returns: `RAX`
- Floating-point returns: `XMM0`
- Small aggregates (up to 16 bytes): returned in registers (`RAX`/`RDX` or `XMM0`/`XMM1` depending on classification)
- Larger aggregates: returned via hidden pointer (caller-allocated storage, pointer passed as implicit first argument)

Register preservation

- Callee-saved GPRs: `RBX, RBP, R12, R13, R14, R15`
- Caller-saved GPRs: `RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11`
- Vector registers: `XMM0--XMM15` and `YMM0--YMM15` are caller-saved on System V AMD64.
- `RSP` must be restored to its incoming value before `ret`.

Stack discipline and alignment

- Stack grows downward; `call` pushes an 8-byte return address.
- **Before every `call`**, the caller must ensure `RSP` is **16-byte aligned**.
- On callee entry (after the return address is pushed): typically `RSP % 16 == 8`.
- Red zone: 128 bytes below `RSP` is usable mainly by leaf functions; never treat it as safe across calls.

13.2 Safe Calling Convention Practices

The practices below reduce risk of hidden ABI bugs and make debugging predictable.

13.2.1 Practice 1: Establish a consistent prologue/epilogue

Use a clear frame strategy and preserve only what you touch.

```
.intel_syntax noprefix
.global disciplined_nonleaf
disciplined_nonleaf:
    push rbp
    mov  rbp, rsp

    push rbx                 # # preserve callee-saved used reg
    sub  rsp, 32             # # locals/spills/padding (example)

    # # ... body ...
    # # If calling other functions, ensure call-site alignment holds.

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

13.2.2 Practice 2: Audit alignment at every call site

A single `push` before a call is enough to violate alignment if not compensated.

```
.intel_syntax noprefix
```

```
.extern callee

.global aligned_call_pattern
aligned_call_pattern:
    push rbx
    sub  rsp, 8                # # compensate to keep 16-byte alignment
     ↪  before call
    call callee
    add  rsp, 8
    pop  rbx
    ret
```

13.2.3 Practice 3: Preserve caller-saved registers only when needed

If you need a volatile value after a call, save it explicitly.

```
.intel_syntax noprefix
.extern callee

.global caller_keeps_r10
caller_keeps_r10:
    mov  r10, 0xABCDEF

    push r10                   # # save volatile
    call callee
    pop  r10                   # # restore

    mov  rax, r10
    ret
```

13.2.4 Practice 4: Treat XMM/YMM as volatile across calls

If your caller relies on `xmmN` surviving, save it.

```
.intel_syntax noprefix
.extern callee

.global caller_keeps_xmm0
caller_keeps_xmm0:
    sub  rsp, 16
    movdqu [rsp], xmm0
    call callee
    movdqu xmm0, [rsp]
    add  rsp, 16
    ret
```

13.2.5 Practice 5: Keep cross-language boundaries "flat"

Prefer C-linkage style boundaries and explicit pointer+length protocols.

```
.intel_syntax noprefix
.global api_sum_bytes
# long api_sum_bytes(const unsigned char* p, long n)
# SysV: p=RDI, n=RSI
api_sum_bytes:
    test rdi, rdi
    je   .bad
    test rsi, rsi
    jle  .bad

    xor eax, eax              # # sum in RAX
```

```
.loop:
    cmp rsi, 0
    je  .done
    movzx edx, byte ptr [rdi]
    add rax, rdx
    inc rdi
    dec rsi
    jmp .loop
.done:
    ret
.bad:
    mov eax, -1
    ret
```

13.3 Common Mistakes to Avoid

13.3.1 Mistake 1: Unbalanced stack adjustments

```
.intel_syntax noprefix
.global bug_unbalanced
bug_unbalanced:
    sub rsp, 16
    # # ... work ...
    ret                         # # BUG: stack not restored
```

13.3.2 Mistake 2: Clobbering callee-saved registers

```
.intel_syntax noprefix
.global bug_clobber_rbx
```

```
bug_clobber_rbx:
    mov rbx, 7                  # # BUG: RBX must be preserved
    ret
```

13.3.3 Mistake 3: Assuming argument registers survive calls

```
.intel_syntax noprefix
.extern callee

.global bug_assume_rdi_survives
bug_assume_rdi_survives:
    mov rdi, 77
    call callee                 # # callee may clobber RDI
    mov rax, rdi                # # BUG
    ret
```

13.3.4 Mistake 4: Misaligned call sites

```
.intel_syntax noprefix
.extern callee

.global bug_misaligned_call
bug_misaligned_call:
    push rbx
    call callee                 # # BUG: may violate 16-byte alignment
    pop  rbx
    ret
```

13.3.5 Mistake 5: Using red zone across calls

```
.intel_syntax noprefix
.extern callee

.global bug_redzone_across_call
bug_redzone_across_call:
    mov qword ptr [rsp - 8], 123    # # stored in red zone
    call callee                     # # callee may overwrite below its
     ↪  RSP
    mov rax, qword ptr [rsp - 8]    # # BUG: value not reliable
    ret
```

13.3.6 Mistake 6: Wrong aggregate return assumption

```
.intel_syntax noprefix
# BUG pattern (conceptual):
# - caller expects a struct returned in RAX/RDX
# - callee actually uses hidden pointer return (or vice versa)
# This mismatch corrupts memory or returns garbage.
```

Final Checklist (Print and Audit)

- Before every call: `RSP` is 16-byte aligned.
- After function return: `RSP` equals its entry value.
- Callee preserves `RBX, RBP, R12--R15` if used.
- Caller saves volatile registers it needs after calls (including `XMM/YMM`).

- Integer/pointer args in `RDI, RSI, RDX, RCX, R8, R9`; FP args in `XMM0--XMM7`.
- Returns: scalars in `RAX` / `XMM0`; aggregates follow the 16-byte vs hidden-pointer rule.
- Do not overwrite `[rsp]` or `[rbp+8]` (return address).
- Use red zone only for leaf temporaries; never rely on it across calls.
- Keep cross-language boundaries flat: C linkage, explicit sizes, no exceptions, no varargs.

References

Architecture Manuals

The contents of this booklet are grounded in the authoritative architectural documentation for the x86-64 platform. These materials define the instruction set behavior, register semantics, stack operation, and control-flow mechanics that the ABI relies on.

- x86-64 instruction execution model, including `call`, `ret`, stack behavior, and register effects.
- General-purpose register and SIMD register architecture (GPRs, XMM/YMM).
- Memory addressing rules, alignment requirements, and calling-related side effects.
- Architectural guarantees required for correct ABI implementation in user-space code.

These manuals form the non-negotiable foundation for understanding how ABI rules map onto real hardware behavior.

ABI Specifications

This booklet follows the official System V AMD64 ABI specification, which defines the binary interface contract between independently compiled translation units.

Core specification areas used throughout this booklet include:

- Register-based argument passing for integer and floating-point arguments.
- Stack layout rules and 16-byte alignment requirements.
- Caller-saved and callee-saved register classification.
- Aggregate classification and return-value handling.
- Hidden pointer mechanism for large return objects.
- Variadic function handling and register save area concepts.

All calling convention rules, stack discipline constraints, and preservation responsibilities described in this booklet are derived directly from this specification.

Compiler Calling Convention Documentation

Modern compilers implement the System V AMD64 ABI with strict conformance, but also expose implementation-specific behavior that is essential for low-level debugging and interoperability.
This booklet aligns with documented compiler behavior in the following areas:

- Register allocation and argument lowering for C and C++ frontends.
- Stack frame generation, prologue/epilogue patterns, and frame pointer usage.
- Inline assembly constraints and clobber rules.
- ABI interactions with optimization levels and inlining.
- Debugging and unwinding expectations tied to correct ABI compliance.

Compiler documentation is treated as a confirmation layer: it demonstrates how the ABI rules are realized in practice, not as an alternative definition of the ABI itself.

Cross-References to Other Booklets in This Series

This booklet is part of the CPU Programming Series and is designed to be read alongside earlier and later volumes. Key conceptual dependencies and continuations include:

- **Booklet 01 — How a CPU Executes Instructions** Foundation for understanding control flow, instruction retirement, and call/return mechanics.
- **Booklet 02 — Registers, Flags, and Data Representation** Required for correct reasoning about register usage, data width, sign extension, and flag side effects.
- **Booklet 03 — The Stack & Calling Conventions (ABI Foundations)** Introduces stack concepts, frame structure, and cross-architecture ABI principles.
- **Booklet 04 — Memory, Caches, and the Cost of Access** Provides essential background for stack memory behavior, alignment costs, and performance implications.
- **Booklet 08 — Windows x64 ABI: Calling Convention Differences That Break Code** Direct comparison volume highlighting incompatibilities between System V AMD64 and Windows x64 ABIs.

Together, these booklets form a coherent progression from CPU fundamentals to ABI-level correctness and cross-platform interoperability.