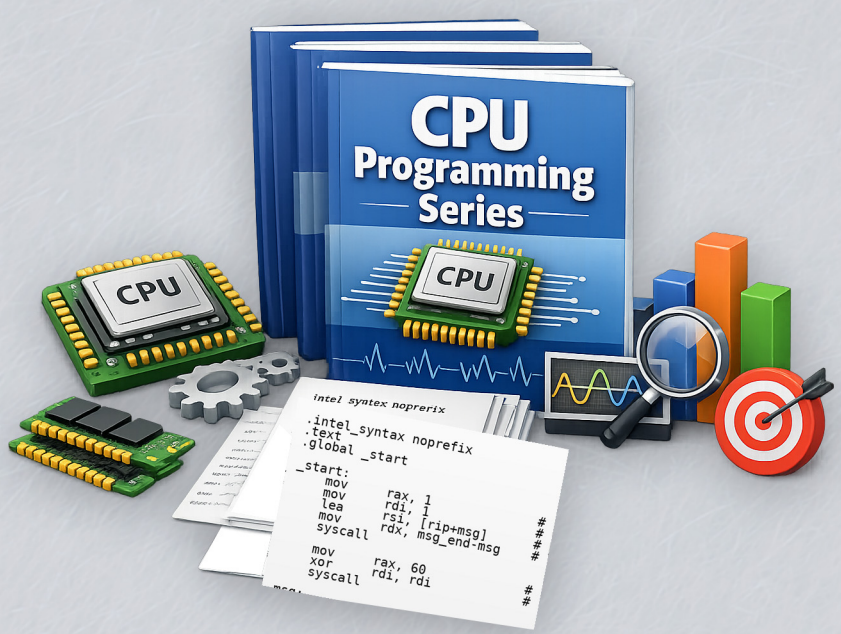
https://simplifycpp.org

CPU Programming Series

RISC-V RV64I Assembly

Bit Addressing & ABI Rules-64



16

Prepared by Ayman Alheraki

CPU Programming Series

RISC-V RV64I

64-Bit Addressing & ABI Rules

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Purpose of This Booklet

This booklet presents a precise and execution-focused explanation of the RISC-V RV64I architecture, concentrating on 64-bit addressing semantics and ABI rules. Its purpose is to establish a correct architectural mental model for reasoning about pointer width, address computation, stack discipline, and function-call correctness on 64-bit RISC-V systems.

The material emphasizes semantic correctness over instruction memorization, showing how 64-bit rules affect real execution, compiler output, and interoperability with C and C++.

Intended Audience

This booklet is intended for systems programmers targeting RV64 platforms, C and C++ developers who need ABI-level understanding, compiler and toolchain developers, and advanced students of computer architecture and low-level programming.

The text assumes an engineering mindset focused on correctness, portability, and long-term maintainability.

Prerequisites and Assumed Knowledge

The reader is expected to understand basic CPU architecture concepts, assembly-level reasoning, and C or C++ data types and calling conventions. Prior familiarity with RV32I is strongly recommended, as RV64I extends the 32-bit execution model rather than replacing it.

How to Read This Booklet

This booklet should be read sequentially. Early chapters establish invariants that later chapters rely on, particularly those related to address width, alignment, and ABI guarantees. Examples are minimal and semantic-driven. For example:

```
/* Load a 64-bit pointer from memory */
ld  x5, 0(x10)

/* Perform 64-bit pointer arithmetic */
addi x5, x5, 16

/* Store a 64-bit value through the pointer */
sd  x6, 0(x5)
```

Each example illustrates architectural behavior rather than assembler convenience.

Relationship to Other Booklets in This Series

This booklet builds directly on the RV32I volume in this series and marks the transition from foundational instruction semantics to full 64-bit system execution. It provides the

architectural groundwork required for later booklets covering RV64 calling conventions, ABI interoperability, compiler- generated assembly analysis, and privilege-level behavior.
Mastery of RV64I addressing and ABI rules is essential before progressing to advanced topics, as violations at this level often compile successfully but fail unpredictably at runtime.

Chapter 1

Introduction

1.1 RV64I in the RISC-V Ecosystem

RV64I is the 64-bit base integer ISA of the RISC-V architecture. It extends RV32I by widening general-purpose registers, addresses, and arithmetic operations to 64 bits while preserving the same fundamental instruction semantics and architectural philosophy.

Within the RISC-V ecosystem, RV64I serves as the **primary foundation for modern general-purpose systems**, including operating systems, language runtimes, and high-performance applications. All higher-level extensions such as virtual memory, privilege modes, atomics, and vector processing assume the execution and data model established by RV64I.

A critical design principle of RISC-V is that RV64I is **not a new ISA**, but a strict superset of RV32I semantics. Instructions behave the same, but with wider registers and addresses. This continuity allows toolchains and mental models developed for RV32I to scale naturally to 64-bit systems.

For example, the following load operation follows the same semantic steps as in RV32I, but operates on 64-bit values and addresses:

```
/* Load a 64-bit value from memory */
ld  x5, 0(x10)
```

The architectural simplicity of RV64I is intentional: complexity is pushed out of the base ISA and into well-defined extensions and ABIs.

1.2 Why 64-Bit Addressing Matters

The transition from 32-bit to 64-bit addressing is not merely about accessing more memory. It fundamentally changes how programs reason about pointers, data layout, and correctness.
In RV64I:

- All general-purpose registers are 64 bits wide
- Pointers are 64 bits
- Address calculations are performed in 64-bit arithmetic
- Stack alignment and ABI rules become stricter

A common source of bugs arises when 32-bit assumptions are silently carried into 64-bit code. For instance, truncating a pointer or assuming 32-bit wrap behavior leads to undefined and often catastrophic failures.
Consider correct 64-bit pointer arithmetic:

```
/* Load base address */
ld   x5, 0(x10)

/* Advance pointer by 32 bytes */
addi x5, x5, 32
```

```
/* Store through updated pointer */
sd   x6, 0(x5)
```

Here, the address computation, register storage, and memory access all occur in 64-bit space. Any attempt to treat the pointer as a 32-bit value would violate architectural guarantees and ABI expectations.
Thus, 64-bit addressing matters because it enforces **discipline, correctness, and scalability** at the architectural level.

1.3 ISA vs ABI: Conceptual Separation

A central theme of this booklet is the strict separation between the **ISA** and the **ABI**. The ISA defines:

- Instruction encodings
- Register behavior
- Memory access semantics
- Arithmetic and control-flow rules

The ABI defines:

- How functions pass arguments and return values
- Which registers must be preserved across calls
- Stack layout and alignment
- How high-level languages map data to machine state

RV64I specifies the execution capabilities of the processor, but it does not dictate how programs cooperate. That cooperation is enforced by the ABI.
For example, the ISA allows any register to be modified, but the ABI imposes rules on which registers must be preserved across a function call:

```
/* Function prologue preserving callee-saved register */
addi sp, sp, -16
sd   x8, 8(sp)

/* Function body */
add  x10, x10, x11

/* Function epilogue restoring register */
ld   x8, 8(sp)
addi sp, sp, 16
ret
```

This code is not required by the ISA to be written this way, but it is required by the ABI for correctness and interoperability.
Understanding this separation is essential. ISA violations cause immediate execution failure. ABI violations often compile and run, but fail silently when programs interact.
This booklet treats **ISA rules as immutable physics** and **ABI rules as non-negotiable contracts**—both must be respected to write correct RV64 code.

Chapter 2

From RV32I to RV64I

2.1 Architectural Continuity

RV64I is a disciplined extension of RV32I. The architectural philosophy and core execution model remain unchanged:

- Load/store design: arithmetic operates on registers, not directly on memory
- Fixed 32-bit base instruction encoding
- 32 integer registers `x0--x31`, with `x0` hard-wired to zero
- Explicit control-flow (branches/jumps/calls) with PC-relative immediates

What RV64I changes is the width of the architectural state:

- Integer register width becomes 64 bits (XLEN = 64)
- Addresses and pointers become 64 bits

- The default integer computation domain becomes 64-bit

This means the RV32I mental model remains valid; the difference is that every "machine word" is now 64-bit, and pointer correctness becomes a primary correctness constraint.

```
/* Same mental model in both RV32I and RV64I: load -> compute ->
↪  store */

/* RV64I: 64-bit data and 64-bit addressing */
ld   x5, 0(x10)       /* x5 = *(uint64_t*)x10 */
add  x6, x5, x11      /* x6 = x5 + x11 */
sd   x6, 8(x10)       /* *(uint64_t*)(x10 + 8) = x6 */
```

2.2 Instruction Compatibility

RV64I preserves the meanings of RV32I instructions while widening registers. In addition, RV64I introduces explicit 32-bit "word" operations to support mixed-width computation.

64-bit operations vs 32-bit word operations

A key rule: word operations compute using the low 32 bits and then **sign-extend** the result to 64 bits.

```
/* 64-bit add: full XLEN width */
add   x5, x6, x7

/* 32-bit add (word): (x6[31:0] + x7[31:0]) then sign-extend to 64 */
addw  x5, x6, x7
```

Immediate arithmetic has the same split:

```
/* 64-bit add immediate */
addi  x5, x5, 1

/* 32-bit add immediate (word) then sign-extend */
addiw x5, x5, 1
```

Load extension rules matter more in RV64I

RV64I makes extension behavior explicit and critical. A 32-bit load into a 64-bit register may sign-extend or zero-extend depending on the instruction.

```
/* Sign-extend 32-bit value to 64-bit register */
lw   x5, 0(x10)

/* Zero-extend 32-bit value to 64-bit register */
lwu  x6, 4(x10)

/* 64-bit load (no extension; full 64-bit value loaded) */
ld   x7, 8(x10)
```

These rules are a compatibility bridge: they allow RV64 code to faithfully manipulate 32-bit data while remaining fully 64-bit correct.

2.3 Practical Implications of the 64-Bit Transition

The transition to RV64 affects correctness in three recurring areas:

- Pointer width and address calculation discipline
- Data layout and alignment under a 64-bit ABI
- Mixed-width arithmetic and extension bugs

Pointer width and truncation hazards

In RV64, pointers are 64-bit. Any accidental truncation to 32 bits destroys high address bits and typically leads to invalid memory accesses.
Correct: keep pointers in full 64-bit registers and use `ld/sd` for pointer-sized objects.

```
/* Load a 64-bit pointer from a table */
ld   x5, 0(x10)      /* x5 = *(uint64_t*)x10 */

/* Use it as an address (still 64-bit) */
ld   x6, 0(x5)       /* x6 = *(uint64_t*)x5 */
```

Incorrect patterns often appear when code treats an address like a 32-bit integer value. A typical hazard is reading a pointer with a 32-bit load:

```
/* WRONG: reads only 32 bits of a 64-bit pointer */
lw   x5, 0(x10)      /* truncates pointer then sign-extends */
```

ABI-driven alignment and stack discipline

On RV64 systems, the ABI typically requires stricter stack alignment (commonly 16-byte aligned at call boundaries). This affects every non-leaf function and any function that saves registers or allocates locals.
Example: minimal non-leaf function preserving a callee-saved register with a 16-byte aligned frame.

```
/* Non-leaf function: preserves x8 and calls helper */
addi sp, sp, -16
sd   x8, 8(sp)       /* save callee-saved */

add  x8, x10, x11    /* use x8 as a local temporary */
```

```
bl   helper

ld   x8, 8(sp)        /* restore */
addi sp, sp, 16
ret
```

Even when code appears to work, violating alignment rules can break:

- code that uses aligned stack accesses
- compiler assumptions for spills and local objects
- debugging/unwinding conventions

Mixed-width arithmetic: choose the width intentionally

RV64I makes it easy to accidentally mix 32-bit intent with 64-bit operations. When the algorithm is truly 32-bit (e.g., hashing, checksums, 32-bit counters), use `addw/addiw` and the correct load extension to preserve semantics.
Example: a 32-bit counter in memory that must behave modulo 2^{32}:

```
/* x10 points to a 32-bit counter */
lwu  x5, 0(x10)       /* zero-extend: treat counter as uint32_t */
addiw x5, x5, 1       /* 32-bit increment, then sign-extend to 64 */
sw   x5, 0(x10)       /* store low 32 bits back */
```

If you instead used `lw` (sign-extend) or `addi` (64-bit add), you can silently change the program meaning, especially near the high bit.
Example: signed 32-bit value that must preserve sign when widened:

```
/* x10 points to a signed int32_t */
lw   x5, 0(x10)       /* sign-extend int32_t to 64 */
```

```
addiw x5, x5, -1      /* 32-bit arithmetic with defined sign behavior
 ↪  */
sw    x5, 0(x10)
```

Practical RV64 programming is therefore not only “64-bit everywhere”; it is about making width explicit and aligning ISA behavior with ABI contracts and language-level intent.

Chapter 3

RV64I Register Model

3.1 General-Purpose Registers

RV64I defines 32 integer general-purpose registers `x0--x31`. Each register holds a 64-bit value (XLEN = 64). The register file is uniform: most instructions can use any integer register as a source or destination.
A defining architectural invariant is:

- `x0` is hard-wired to zero. Reads always return 0; writes are ignored.

This enables common idioms without dedicated instructions:

```
/* Clear a register */
add  x5, x0, x0      /* x5 = 0 */

/* Move between registers */
add  x6, x5, x0      /* x6 = x5 */
```

```
/* Compare against zero using branch */
bne  x7, x0, nonzero /* if (x7 != 0) goto nonzero */
```

Because RV64I is a pure load/store ISA, memory operands do not exist in ALU instructions. Any computation that involves memory always follows the same pattern:

```
/* load -> compute -> store */
ld   x5, 0(x10)       /* x5 = *(uint64_t*)x10 */
add  x6, x5, x11      /* x6 = x5 + x11 */
sd   x6, 8(x10)       /* *(uint64_t*)(x10 + 8) = x6 */
```

3.2 Register Width and Naming

Architectural width

In RV64I, integer registers are 64-bit wide. Many instructions naturally operate on 64-bit values (e.g., `add`, `sub`, `and`, `or`, shifts, comparisons). Immediate adds and address calculations also occur in 64-bit arithmetic:

```
/* 64-bit arithmetic */
add   x5, x6, x7
sub   x8, x8, x9
addi  x10, x10, 32
```

RV64I additionally provides explicit 32-bit *word* operations. These compute using the low 32 bits and then sign-extend the result to 64 bits. This is essential for preserving 32-bit semantics inside a 64-bit machine:

```
/* 32-bit arithmetic inside 64-bit registers */
addw  x5, x6, x7      /* (x6[31:0] + x7[31:0]) then sign-extend */
addiw x5, x5, 1       /* 32-bit add immediate then sign-extend */
```

Naming conventions

Architecturally, registers are named `x0--x31`. In practice, ABI documentation and toolchains often use conventional names to reflect roles in function calls and stack discipline:

- `ra` for return address register
- `sp` for stack pointer
- `gp` for global pointer (platform-defined usage)
- `tp` for thread pointer (platform-defined usage)
- `a0--a7` for argument/return registers
- `t0--t6` for temporaries (caller-saved)
- `s0--s11` for saved registers (callee-saved)

This booklet uses `xN` in core ISA explanations and introduces ABI names when discussing calling convention rules.

3.3 ABI-Oriented Register Roles

The ISA allows any register to be used freely, but the ABI imposes contracts so separately compiled code can interoperate. The most important ABI-driven roles are:

Special-purpose registers

- `sp`: stack pointer. Must remain aligned per ABI at call boundaries.
- `ra`: return address. Set by call instructions; used by `ret`.

```
/* A typical call/return flow */
bl   callee          /* ra = return address; jump to callee */
ret                 /* jump to ra */
```

Argument and return registers

Most RV64 ABIs use `a0--a7` to pass up to 8 integer/pointer arguments. Return values typically use `a0` (and `a1` for wider/multiple returns, depending on type and ABI rules).

```
/* Conceptual: a0,a1 hold args; return in a0 */
add  x10, x10, x11   /* a0 = a0 + a1 */
ret
```

Caller-saved vs callee-saved

ABIs partition registers into:

- Caller-saved: the caller must assume they may be clobbered by a call.
- Callee-saved: if the callee uses them, it must save/restore them.

A minimal non-leaf function that uses a callee-saved register and calls another function must preserve it:

```
/* Uses x8 (commonly s0) as a stable local across a call */
addi sp, sp, -16
sd   x8, 8(sp)       /* save callee-saved */

add  x8, x10, x11    /* x8 is a local persistent value */
bl   helper          /* helper may clobber caller-saved regs */

add  x10, x8, x0     /* return value in a0 (x10) */
```

```
ld   x8, 8(sp)        /* restore */
addi sp, sp, 16
ret
```

In contrast, a leaf function that uses only caller-saved temporaries often needs no stack frame:

```
/* Leaf function: uses temporaries only, no calls */
add  x5, x10, x11     /* temp = a0 + a1 */
xor  x10, x5, x12     /* a0 = temp ^ a2 */
ret
```

Stable values across calls

If a value must survive across a function call, you must either:

- place it in a callee-saved register (and preserve it if you are the callee), or
- spill it to the stack before the call and reload after.

Stack-spill example preserving a live value across a call:

```
/* Preserve a live 64-bit value across a call */
addi sp, sp, -16
sd   x5, 8(sp)        /* save live value */

bl   helper

ld   x5, 8(sp)        /* restore live value */
addi sp, sp, 16
ret
```

The key principle is simple:

- The ISA defines what is possible.
- The ABI defines what is safe for separately compiled code to assume.

This booklet treats ABI register roles as non-negotiable contracts required for correctness, interoperability, and reliable debugging on RV64 systems.

Chapter 4

RV64 Data and Memory Model

4.1 Integer and Pointer Sizes

In RV64I, the architectural integer register width is 64 bits (XLEN = 64), which makes 64-bit integer and pointer operations the natural default. In practice, however, real programs use a mixture of widths: 8/16/32/64-bit integers in memory, with well-defined extension rules when loaded into 64-bit registers.

Register width vs object width

A register is always 64-bit wide, but an object in memory may be smaller. Loads define how the smaller object is extended to 64 bits:

```
/* x10 points to memory containing various integer widths */
lb   x5, 0(x10)      /* int8_t   -> sign-extend to 64 */
lbu  x6, 1(x10)      /* uint8_t  -> zero-extend to 64 */
lh   x7, 2(x10)      /* int16_t  -> sign-extend to 64 */
lhu  x8, 4(x10)      /* uint16_t -> zero-extend to 64 */
```

```
lw   x9, 8(x10)       /* int32_t  -> sign-extend to 64 */
lwu  x11, 12(x10)     /* uint32_t -> zero-extend to 64 */
ld   x12, 16(x10)     /* int64_t/uint64_t -> full 64-bit */
```

Stores write the low bits of the register to memory, according to width:

```
/* Store low bits of a register into memory */
sb   x5, 0(x10)       /* store low 8 bits */
sh   x6, 2(x10)       /* store low 16 bits */
sw   x7, 8(x10)       /* store low 32 bits */
sd   x8, 16(x10)      /* store full 64 bits */
```

Pointers in RV64

Pointers are 64-bit values in RV64. All address calculations should be performed in 64-bit registers, and pointer-sized objects in memory should be loaded/stored with `ld/sd`.

```
/* Load a 64-bit pointer from a pointer table */
ld   x5, 0(x10)       /* x5 = *(uint64_t*)x10 */

/* Dereference pointer */
ld   x6, 0(x5)        /* x6 = *(uint64_t*)x5 */

/* Store pointer back */
sd   x5, 8(x10)
```

A frequent RV64 bug is accidentally treating a pointer like a 32-bit value:

```
/* WRONG: reads only 32 bits of a 64-bit pointer */
lw   x5, 0(x10)       /* truncates pointer then sign-extends */
```

4.2 LP64 Data Model

Most RV64 Unix-like environments use the LP64 data model, whose defining properties are:

- `long` is 64-bit
- pointers are 64-bit
- `int` remains 32-bit

This is not an ISA rule; it is an ABI/platform convention that controls C/C++ type sizes and calling convention behavior. The practical consequence is that code must never assume `sizeof(long) == 4` on RV64 systems.

Type-size consequences

Typical LP64 expectations:

- `char` = 8
- `short` = 16
- `int` = 32
- `long` = 64
- `long long` = 64
- pointer = 64

Assembly implications: a C `int` in memory is usually manipulated with `lw/sw`, while a C `long` or pointer is manipulated with `ld/sd`.
Example: `int` and `long` variables stored adjacent in memory.

```
/* Layout (conceptual):
   [x10+0]  int32_t  i
   [x10+8]  int64_t  L   (aligned)
*/

/* i = i + 1 (32-bit) */
lw   x5, 0(x10)       /* sign-extend int32_t */
addiw x5, x5, 1       /* 32-bit add then sign-extend */
sw   x5, 0(x10)

/* L = L + 1 (64-bit) */
ld   x6, 8(x10)
addi x6, x6, 1
sd   x6, 8(x10)
```

Note the deliberate use of `addiw` for 32-bit semantics and `addi` for 64-bit semantics.

4.3 Alignment and Padding Rules

RV64 systems are sensitive to alignment because:

- ABIs require specific alignment for stack and aggregate layout
- compilers assume alignment for efficient code generation
- some platforms may restrict or penalize misaligned accesses

Alignment is a property of an object in memory, not a property of registers. A correct RV64 mental model is: **alignment rules are enforced at the ABI and code-generation level, even if the ISA can sometimes tolerate misalignment.**

Natural alignment guideline

A practical rule used by compilers and ABIs is natural alignment:

- 1-byte objects aligned to 1
- 2-byte objects aligned to 2
- 4-byte objects aligned to 4
- 8-byte objects aligned to 8

Thus, 64-bit loads/stores (`ld/sd`) are expected to target 8-byte aligned addresses in well-formed ABI code.

Struct padding example (conceptual LP64 layout)

Consider a common mixed-width record:

```
/* Conceptual C layout under LP64:
   struct S {
     uint8_t  a;    // offset 0
     uint32_t b;    // offset 4  (padding inserted)
     uint64_t c;    // offset 8
   };
   sizeof(S) = 16
*/
```

The padding between `a` and `b` exists to keep `b` aligned to 4, and `c` aligned to 8. This affects how assembly code accesses fields.
Assuming `x10` points to an instance of `S`:

```
/* a (uint8_t) at offset 0 */
lbu  x5, 0(x10)

/* b (uint32_t) at offset 4 */
lwu  x6, 4(x10)

/* c (uint64_t) at offset 8 */
ld   x7, 8(x10)
```

If you incorrectly assumed packed layout and used offsets 1 and 5, you would read unrelated bytes and silently corrupt meaning.

Stack alignment and local storage

A typical RV64 ABI requires the stack pointer to be aligned at call boundaries (commonly 16-byte aligned). Local allocations must preserve this property.
Example: allocate 32 bytes of locals while maintaining alignment:

```
/* Allocate an aligned stack frame */
addi sp, sp, -32      /* keep sp aligned */
sd   x8, 24(sp)       /* save callee-saved if needed */
```

Restore:

```
/* Deallocate stack frame */
ld   x8, 24(sp)
addi sp, sp, 32
ret
```

Addressing scaled by element size

Alignment and size also determine correct indexing. For a 64-bit array, the element stride is 8 bytes:

```
/* x10 = base pointer to uint64_t array
   x11 = index i
   Goal: load array[i] into x5
*/
slli x12, x11, 3     /* offset = i * 8 */
add  x12, x10, x12   /* address = base + offset */
ld   x5, 0(x12)
```

For a 32-bit array, the stride is 4 bytes:

```
/* base = uint32_t*, stride = 4 */
slli x12, x11, 2     /* offset = i * 4 */
add  x12, x10, x12
lwu  x5, 0(x12)      /* unsigned load preserves uint32_t meaning */
```

The main principle is consistent across the booklet:

- Width determines extension rules and instruction choice.
- ABI determines layout, alignment, and interoperability guarantees.
- Correct RV64 code makes both explicit.

Chapter 5

64-Bit Addressing Semantics

5.1 Virtual Address Space Overview

RV64I provides 64-bit integer registers and 64-bit virtual addresses in the architectural model, but real implementations typically use fewer virtual address bits than 64. The exact number of implemented virtual address bits is a platform and implementation choice and is reflected in the virtual memory subsystem and operating system configuration.

Even when fewer address bits are implemented, software must treat pointers as 64-bit values and must follow the platform's canonical-address rules. The safest practical rule is:

- Treat every address as a full 64-bit value.
- Never truncate pointers.
- Form addresses only through well-defined pointer arithmetic.

User and kernel regions (conceptual)

Most operating systems logically partition virtual memory into regions reserved for user space and kernel space. The boundaries are platform- and OS-defined, but the concept is stable:

- User space: where application pointers are expected to fall
- Kernel space: privileged mappings, not accessible in user mode

A practical consequence for low-level code (toolchains, runtimes, kernels) is that an address may be considered invalid even if it "fits" in 64 bits, because it violates canonical form or region constraints.

Pointer-sized loads/stores

In RV64, pointers are 64-bit objects and should be accessed with `ld/sd`:

```
/* Load/store a pointer-sized value */
ld   x5, 0(x10)      /* x5 = *(uint64_t*)x10   (pointer) */
sd   x5, 8(x10)      /* *(uint64_t*)(x10+8) = x5 */
```

Loading a pointer with a 32-bit load is almost always a bug:

```
/* WRONG: truncates a 64-bit pointer */
lw   x5, 0(x10)      /* reads low 32 bits then sign-extends */
```

5.2 Canonical Addressing Rules

RV64 systems typically impose a canonical-address requirement: only a subset of the 64-bit space is considered a valid virtual address, and the remaining high bits must match the sign bit of the implemented virtual address width.

Conceptually, if the platform implements N virtual address bits (where $N < 64$), then bits $63 : N$ must be a sign-extension of bit $N - 1$. Addresses that violate this rule are *non-canonical* and are treated as invalid by the MMU and OS.
This rule ensures that different, non-equal 64-bit values cannot alias the same virtual address, and it provides a clean, consistent meaning for pointers even when the implementation uses fewer than 64 virtual bits.

Canonicalization patterns (conceptual)

If you have an address-like value that is known to represent a valid virtual address in the implemented width, canonicalization typically means sign-extending the top implemented bit.
A common conceptual technique is:

- Shift left so the would-be sign bit moves into bit 63.
- Arithmetic shift right back to sign-extend.

```
/* Conceptual canonicalization pattern for an N-bit virtual address
   x5 holds an address-like value with meaningful low N bits
   shift = 64 - N
*/
slli x5, x5, shift
srai x5, x5, shift    /* arithmetic shift right: sign-extend */
```

Notes:

- This is a conceptual pattern; real software usually relies on OS and compiler guarantees that pointers are already canonical.
- Canonicalization is not a substitute for validating that the address actually refers to a mapped, accessible region.

Detecting non-canonical addresses (conceptual)

A conceptual validity check compares an address against its canonicalized form:

```
/* Conceptual: if canonicalize(x5) != x5 then non-canonical */
add  x6, x5, x0       /* x6 = original */
slli x5, x5, shift
srai x5, x5, shift
bne  x5, x6, noncanonical
```

5.3 Address Calculation and Overflow

Address calculation in RV64 uses 64-bit arithmetic in the integer registers. The ISA does not provide a separate "address arithmetic" unit; effective addresses are computed exactly as integer expressions.

Effective address computation for loads/stores

A load/store uses a base register plus a signed immediate. The base is 64-bit, and the immediate is sign-extended before addition. The resulting 64-bit sum is the effective virtual address:

```
/* Effective address = x10 + signext(imm12) */
ld   x5, 32(x10)
sd   x6, -16(x10)
```

For larger offsets, you must synthesize the address in registers:

```
/* address = base + large_offset; then access */
li   x12, 4096
add  x12, x10, x12
ld   x5, 0(x12)
```

Array indexing must scale by element size

Correct address computation requires scaling indices by element width:

```
/* uint64_t a[i]: address = base + i*8 */
slli x12, x11, 3      /* x12 = i * 8 */
add  x12, x10, x12    /* x12 = base + i*8 */
ld   x5, 0(x12)
```

```
/* uint32_t b[i]: address = base + i*4 */
slli x12, x11, 2      /* x12 = i * 4 */
add  x12, x10, x12
lwu  x5, 0(x12)       /* preserve uint32_t meaning */
```

Overflow, wraparound, and why it is dangerous

Integer addition in RV64 is modulo 2^{64}. That means address arithmetic can wrap around if the sum exceeds the 64-bit range. The ISA will still produce a 64-bit result, but that result may:

- become non-canonical on systems with canonical addressing rules
- land in a different virtual region (e.g., kernel range)
- produce an unmapped address and fault when dereferenced

Example: unchecked pointer increment in a loop:

```
/* x10 = ptr, x11 = count, stride = 8 */
loop:
ld   x5, 0(x10)
addi x10, x10, 8      /* ptr += 8 (may overflow/wrap if unchecked) */
addi x11, x11, -1
bne  x11, x0, loop
```

Correct low-level code treats pointer arithmetic as a bounded operation: the algorithm must ensure the pointer remains within a valid object/region.

Detecting unsigned overflow (carry) when needed

When implementing allocators, bounds checks, or hardened runtimes, you may need to detect overflow in address calculations. A common unsigned-overflow test is:

- `sum = a + b`
- overflow occurred if `sum < a` (unsigned comparison)

```
/* Detect overflow of (x10 + x11) treating them as unsigned */
add  x12, x10, x11     /* sum */
bltu x12, x10, overflow
```

If overflow is detected, robust code avoids dereferencing the computed address.

Signed overflow is not the model for addresses

Addresses should be treated as unsigned quantities for overflow reasoning. Signed comparisons and signed overflow intuitions are often incorrect for pointer arithmetic and virtual address validity.
A safe rule for RV64 systems programming:

- Use unsigned comparisons for bounds and overflow checks on addresses.
- Keep addresses canonical and within known valid regions.

Chapter 6

Load and Store Semantics

6.1 Load and Store Instructions

RV64I is a load/store architecture: only load instructions read memory into registers, and only store instructions write register values back to memory. Integer ALU instructions never operate directly on memory.

Widths and basic forms

RV64I provides loads and stores for common integer widths. Loads place a value into a 64-bit register; stores write low bits of a register to memory.

- Byte: `lb`, `lbu`, `sb`
- Halfword (16-bit): `lh`, `lhu`, `sh`
- Word (32-bit): `lw`, `lwu`, `sw`
- Doubleword (64-bit): `ld`, `sd`

All load/store effective addresses use the same base+offset form: `(base register) + sign-extended immediate`.

```
/* Base+offset addressing */
lb   x5, 0(x10)
lh   x6, 2(x10)
lw   x7, 4(x10)
ld   x8, 8(x10)

sb   x5, 0(x10)
sh   x6, 2(x10)
sw   x7, 4(x10)
sd   x8, 8(x10)
```

Load -¿ compute -¿ store is the only correct model

```
/* Sum two 64-bit integers from memory and store the result */
ld   x5, 0(x10)       /* x5 = *(uint64_t*)x10 */
ld   x6, 8(x10)       /* x6 = *(uint64_t*)(x10 + 8) */
add  x7, x5, x6       /* x7 = x5 + x6 */
sd   x7, 16(x10)      /* *(uint64_t*)(x10 + 16) = x7 */
```

Scaling offsets for arrays

For element indexing, the offset must be scaled by element size.

```
/* uint64_t a[i] at base x10, index i in x11 */
slli x12, x11, 3      /* offset = i * 8 */
add  x12, x10, x12
ld   x5, 0(x12)
```

```
/* uint32_t b[i] at base x10, index i in x11 */
slli x12, x11, 2     /* offset = i * 4 */
add  x12, x10, x12
lwu  x5, 0(x12)
```

6.2 Sign Extension vs Zero Extension

In RV64, a load of a smaller-than-64-bit object must specify how the value is extended to 64 bits. This is not cosmetic. It changes the meaning of the data.

Rule summary

- `lb`, `lh`, `lw` sign-extend to 64
- `lbu`, `lhu`, `lwu` zero-extend to 64
- `ld` loads a full 64-bit value (no extension step)

Example: same bytes, different meaning

Assume memory contains `0xFF` at `0(x10)`:

```
/* int8_t v = *(int8_t*)p; */
lb   x5, 0(x10)      /* x5 = 0xFFFFFFFFFFFFFFFF (-1) */

/* uint8_t u = *(uint8_t*)p; */
lbu  x6, 0(x10)      /* x6 = 0x00000000000000FF (255) */
```

Assume memory contains `0x80000000` at `0(x10)`:

```
/* int32_t a = *(int32_t*)p; */
lw   x5, 0(x10)       /* x5 = 0xFFFFFFFF80000000 (negative) */

/* uint32_t b = *(uint32_t*)p; */
lwu  x6, 0(x10)       /* x6 = 0x0000000080000000 (positive) */
```

Correct instruction choice for common C/C++ types (LP64)

On most RV64 platforms, `int` is 32-bit and `long` and pointers are 64-bit. This implies:

```
/* C int32_t / int */
lw   x5, 0(x10)       /* signed int */
lwu  x6, 4(x10)       /* unsigned int */

/* C int64_t / long / pointer */
ld   x7, 8(x10)       /* 64-bit signed/unsigned/pointer value */
```

Preserving 32-bit arithmetic intent in RV64

If the algorithm is defined in 32-bit arithmetic (common in hashing, CRC, bit-manipulation, network protocols), keep the semantics explicit using word ops and the correct load extension.

```
/* uint32_t x = *(uint32_t*)p; x = x + 1; */
lwu  x5, 0(x10)       /* zero-extend uint32_t */
addiw x5, x5, 1       /* 32-bit add then sign-extend */
sw   x5, 0(x10)       /* store low 32 bits */
```

For signed 32-bit semantics:

```
/* int32_t y = *(int32_t*)p; y = y - 1; */
lw   x5, 0(x10)       /* sign-extend int32_t */
addiw x5, x5, -1      /* 32-bit arithmetic */
sw   x5, 0(x10)
```

6.3 Unaligned Memory Access

Alignment is an ABI and performance-critical concern. While some hardware may support certain unaligned accesses, software that targets portability and ABI correctness should assume:

- Naturally aligned accesses are the norm and the expectation.
- Unaligned accesses may be slower, trapped, or require emulation, depending on implementation and privilege mode.

Practical guidance for RV64 systems programming:

- Use aligned object layouts for structs and arrays.
- Ensure stack pointer alignment at call boundaries.
- When parsing packed byte streams, use byte loads and reconstruct values.

Aligned access examples

```
/* x10 points to 8-byte aligned storage */
ld   x5, 0(x10)
sd   x5, 8(x10)
```

Packed data parsing: reconstruct safely from bytes

When reading network packets or file formats, data may be packed and unaligned. A robust method is to load bytes and assemble the value.
Example: load a little-endian 32-bit unsigned value from an arbitrary address:

```
/* x10 = pointer to packed bytes */
lbu  x5, 0(x10)       /* b0 */
lbu  x6, 1(x10)       /* b1 */
lbu  x7, 2(x10)       /* b2 */
lbu  x8, 3(x10)       /* b3 */

slli x6, x6, 8
slli x7, x7, 16
slli x8, x8, 24

or   x5, x5, x6
or   x5, x5, x7
or   x5, x5, x8        /* x5 = assembled uint32_t in low 32 bits */
```

If you need the result as a 64-bit unsigned value with zero-extension:

```
/* Zero-extend assembled 32-bit value (optional canonical step) */
slli x5, x5, 32
srli x5, x5, 32
```

Example: load a little-endian 64-bit value from packed bytes:

```
/* x10 = pointer to packed bytes; result in x5 */
lbu  x5, 0(x10)
lbu  x6, 1(x10)
lbu  x7, 2(x10)
lbu  x8, 3(x10)
lbu  x9, 4(x10)
lbu  x11, 5(x10)
lbu  x12, 6(x10)
lbu  x13, 7(x10)
```

```
slli x6,  x6,  8
slli x7,  x7,  16
slli x8,  x8,  24
slli x9,  x9,  32
slli x11, x11, 40
slli x12, x12, 48
slli x13, x13, 56

or   x5, x5, x6
or   x5, x5, x7
or   x5, x5, x8
or   x5, x5, x9
or   x5, x5, x11
or   x5, x5, x12
or   x5, x5, x13      /* x5 = assembled uint64_t */
```

Why this matters

ABI-correct code assumes aligned objects and aligned stack frames. Unaligned access is therefore treated as an exceptional case reserved for packed I/O formats and byte streams. In those cases, reconstruct values explicitly to avoid relying on implementation-specific unaligned behavior.

Chapter 7

Arithmetic and Control Flow

7.1 64-Bit Arithmetic Operations

In RV64I, the default integer arithmetic domain is 64-bit (XLEN = 64). All general-purpose registers hold 64-bit values, and standard arithmetic and logical instructions operate on the full register width unless explicitly restricted.
Common 64-bit arithmetic and logical operations include addition, subtraction, shifts, and bitwise logic:

```
/* 64-bit integer arithmetic */
add   x5, x6, x7     /* x5 = x6 + x7 */
sub   x8, x8, x9     /* x8 = x8 - x9 */
and   x10, x10, x11  /* bitwise AND */
or    x12, x12, x13  /* bitwise OR */
xor   x14, x14, x15  /* bitwise XOR */
```

Shifts operate on 64-bit values, with the shift count taken from the low bits of the shift operand:

```
/* 64-bit shifts */
sll   x5, x6, x7     /* logical left shift */
srl   x8, x8, x9     /* logical right shift */
sra   x10, x10, x11  /* arithmetic right shift */
```

Immediate forms are frequently used for address calculations and loop counters:

```
/* 64-bit add immediate */
addi  x5, x5, 16     /* x5 += 16 */
```

All arithmetic is modulo 2^{64}. The ISA does not trap on overflow. If overflow detection is required, it must be implemented explicitly using comparisons.
Example: detect unsigned overflow of a 64-bit addition:

```
/* unsigned overflow if (a + b) < a */
add   x12, x10, x11
bltu  x12, x10, overflow
```

7.2 Word Instructions (W-Suffix)

RV64I introduces word instructions (with a `W` suffix) to support explicit 32-bit arithmetic within a 64-bit register file. These instructions:

- Operate on the low 32 bits of the source registers
- Produce a 32-bit result
- Sign-extend the result to 64 bits

This behavior is essential for preserving 32-bit language semantics on a 64-bit machine.

Arithmetic word operations

```
/* 32-bit arithmetic with sign extension */
addw  x5, x6, x7     /* (x6[31:0] + x7[31:0]) -> sign-extend */
subw  x8, x8, x9     /* 32-bit subtract */
addiw x10, x10, 1    /* 32-bit add immediate */
```

Word shifts also exist:

```
/* 32-bit shifts, then sign-extend */
sllw  x5, x6, x7
srlw  x8, x8, x9
sraw  x10, x10, x11
```

Why word instructions matter

Using full 64-bit arithmetic when the algorithm is defined in 32-bit terms changes behavior near the sign bit or overflow boundary.
Example: incrementing a 32-bit counter stored in memory:

```
/* Correct 32-bit semantics */
lwu   x5, 0(x10)     /* load uint32_t */
addiw x5, x5, 1      /* 32-bit increment */
sw    x5, 0(x10)
```

Using `addi` instead of `addiw` would turn the operation into a 64-bit increment, silently changing the semantics.

7.3 Branches, Jumps, and Calls

Control flow in RV64I is explicit and orthogonal. There are no implicit condition codes or flags; all decisions are based on register comparisons.

Conditional branches

Branch instructions compare two registers and branch relative to the program counter if the condition is true:

```
/* Signed comparisons */
beq   x5, x6, equal
bne   x5, x6, notequal
blt   x5, x6, less
bge   x5, x6, ge

/* Unsigned comparisons */
bltu  x5, x6, less_u
bgeu  x5, x6, ge_u
```

Branch offsets are PC-relative and limited in range, which encourages short, local control-flow constructs.

Unconditional jumps

Unconditional jumps are performed using `jal` (jump and link) or `jalr` (jump and link register):

```
/* Unconditional PC-relative jump */
jal   x0, target     /* jump without saving return address */

/* Jump via register */
jalr  x0, 0(x5)      /* jump to address in x5 */
```

Using `x0` as the destination discards the link value, making the jump purely unconditional.

Function calls and returns

Function calls are a specialized use of jumps. A call saves the return address in `ra` and transfers control to the callee.

```
/* Function call */
jal    ra, callee

/* Return from function */
ret
```

The `ret` pseudo-instruction expands to a `jalr` that jumps to the address in `ra`.

Control flow with arithmetic

Arithmetic and control flow are often combined in loops:

```
/* Simple countdown loop */
loop:
addi   x10, x10, -1    /* decrement counter */
bne    x10, x0, loop   /* continue while counter != 0 */
```

Because there are no flags, the value produced by arithmetic instructions is directly consumed by branches. This explicit data flow is a core part of the RISC-V design philosophy.

Signed vs unsigned intent

Correct branch selection depends on whether values are signed or unsigned. Using the wrong comparison instruction is a common source of subtle bugs.
Example: bounds check on an array index:

```
/* if (index < size) using unsigned comparison */
bltu  x11, x12, in_range
```

Using a signed branch here would mis-handle values with the high bit set.
The guiding rule for RV64I control flow is:

- Arithmetic defines values.
- Branches interpret values.
- The programmer must make signedness and width explicit.

Chapter 8

RV64 ABI Fundamentals

8.1 ABI Design Goals

An ABI (Application Binary Interface) defines the binary contract that allows separately compiled code to interoperate correctly. On RV64 systems, the ABI binds together:

- how functions pass arguments and return values
- which registers must be preserved across calls
- stack layout and alignment requirements
- object layout rules that affect aggregates and calling convention
- assumptions used by compilers, linkers, debuggers, and runtimes

The RV64 ISA defines what the CPU *can* execute. The ABI defines what compiled code is *allowed to assume* about other compiled code.
In practice, ABI design targets three engineering goals:

Correct interoperability

The ABI ensures that a caller compiled by one compiler can call a callee compiled by another compiler, and both agree on where arguments live, where return values appear, and what state is preserved.

```
/* Caller: place args, call, read return
   Convention: a0-a7 carry integer/pointer args, return in a0
*/

/* a0 = x10, a1 = x11 already set by caller */
jal  ra, add2
/* result returned in a0 (x10) */
```

Performance and predictability

Passing small scalars in registers is faster than passing on the stack. A fixed register convention enables aggressive optimization while preserving predictable call boundaries.

Toolchain compatibility

Debuggers, unwinders, exception mechanisms, profilers, and sanitizers rely on ABI-defined stack discipline and register saving rules. ABI violations may run "fine" until tooling or optimization exposes undefined assumptions.

8.2 Register Classification

The ABI classifies integer registers by role in the calling convention. The ISA exposes `x0--x31`; the ABI assigns meaning to these registers to support portable binaries.

Core special registers

- `ra` (return address): holds the return PC after a call
- `sp` (stack pointer): points to the current stack frame
- `gp` (global pointer): platform-defined use for global access model
- `tp` (thread pointer): platform-defined use for thread-local access

```
/* Call/return mechanics */
jal  ra, callee
ret
```

Argument and return registers

Most RV64 ABIs use `a0--a7` for integer and pointer arguments. Return values are placed in `a0` (and `a1` when needed by the ABI rules).
Example: a leaf function returning `a0 + a1`:

```
/* int64_t add2(int64_t x, int64_t y) */
add  x10, x10, x11   /* a0 = a0 + a1 */
ret
```

Example: passing more than 8 integer/pointer arguments forces overflow to the stack by ABI rules (conceptual illustration of the call boundary):

```
/* a0-a7 hold first 8 args; remaining args are placed by caller in
 ↪  outgoing stack area */
addi sp, sp, -32     /* reserve outgoing area (example size) */
sd   x18, 0(sp)      /* 9th arg (example) */
sd   x19, 8(sp)      /* 10th arg (example) */
jal  ra, callee10
addi sp, sp, 32
```

Caller-saved vs callee-saved

The ABI partitions registers into:

- caller-saved: may be clobbered by the callee; caller must save if needed
- callee-saved: must be preserved by the callee if it uses them

Typical usage model:

- temporaries (`t*`) are caller-saved
- saved registers (`s*`) are callee-saved

Example: caller saves a temporary across a call:

```
/* Preserve a caller-owned temporary (x5) across a call */
addi sp, sp, -16
sd   x5, 8(sp)
jal  ra, helper
ld   x5, 8(sp)
addi sp, sp, 16
```

Example: callee preserves a callee-saved register it uses:

```
/* Callee uses x8 (commonly s0) and must preserve it */
addi sp, sp, -16
sd   x8, 8(sp)

add  x8, x10, x11    /* x8 holds a stable local value */
bl   helper

add  x10, x8, x0     /* return via a0 */
```

```
ld   x8, 8(sp)
addi sp, sp, 16
ret
```

Stack alignment is part of register discipline

Stack pointer alignment at call boundaries is an ABI rule. Frames must be sized and adjusted to preserve alignment.

```
/* Aligned stack frame example (commonly 16-byte alignment) */
addi sp, sp, -32
sd   x8, 24(sp)
sd   x1, 16(sp)      /* ra if needed */
```

8.3 ABI Stability and Portability

ABI stability means that binaries compiled at different times, by different toolchains, or as separate libraries can still link and run together, as long as they follow the same ABI contract. Portability means the same source can be compiled into correct binaries across systems that implement the same ABI.

What is stable vs what can vary

Stable under an ABI:

- argument locations and calling convention rules
- callee-saved vs caller-saved guarantees
- stack alignment requirements at call boundaries

- object layout rules (subject to ABI and language rules)

May vary across platforms even on RV64:

- exact virtual address width and canonical-address constraints
- OS-specific thread-local and global-pointer models
- system-call conventions and privilege interfaces
- library choices and runtime policies

Portability rule for low-level code

Low-level code should treat the ABI as a contract stronger than the ISA:

- ISA rules define execution.
- ABI rules define correctness across compilation units.

If you violate ISA rules, you fault immediately. If you violate ABI rules, you often get delayed failures: incorrect returns, corrupted locals, misinterpreted arguments, or broken debugging and unwinding.

Practical portability checklist

- Pass integer/pointer arguments in `a0--a7`; overflow to stack only when required.
- Preserve callee-saved registers you modify; assume caller-saved are clobbered by calls.
- Maintain required stack alignment at every call boundary.
- Use correct load extension (`lw` vs `lwu`) to match C signedness.
- Use `ld/sd` for pointer-sized objects; never truncate pointers.

Example: a small ABI-correct non-leaf function that uses a stable local across a call and returns it:

```
/* int64_t f(int64_t x, int64_t y) { t = x+y; helper(); return t; }
 ↪  */
addi sp, sp, -32
sd   x8, 24(sp)       /* save callee-saved */
sd   x1, 16(sp)       /* save ra */

add  x8, x10, x11     /* t in x8 (stable across call) */
jal  ra, helper
add  x10, x8, x0      /* return t in a0 */

ld   x1, 16(sp)
ld   x8, 24(sp)
addi sp, sp, 32
ret
```

This pattern remains correct and portable as long as the ABI contract is respected, regardless of compiler version or optimization level.

Chapter 9

Calling Convention and Stack Discipline

9.1 Argument Passing Rules

On RV64 systems, the ABI defines how arguments are delivered to a callee. The most common rule set is:

- Integer and pointer arguments are passed in `a0--a7` (x10–x17).
- If there are more than 8 integer/pointer arguments, the remaining ones are passed on the stack in the caller's outgoing argument area.
- The callee may freely clobber caller-saved registers; the caller must preserve values it needs across a call.

Up to 8 integer/pointer arguments

Example: a function with 4 integer arguments:

```
/* int64_t f(int64_t a, int64_t b, int64_t c, int64_t d)
```

```
   a0=a, a1=b, a2=c, a3=d
*/
add  x10, x10, x11   /* a0 = a0 + a1 */
add  x10, x10, x12   /* a0 = a0 + a2 */
add  x10, x10, x13   /* a0 = a0 + a3 */
ret
```

More than 8 arguments: stack overflow area (conceptual)

When integer/pointer arguments exceed `a0--a7`, the caller places the remaining arguments into memory at known locations relative to the stack pointer at call time. The callee then loads them.

Example: 10 arguments, where args 9 and 10 are on the stack:

```
/* Caller side (conceptual):
   a0-a7 = args 1..8
   stack = args 9..10
*/
addi sp, sp, -32
sd   x18, 0(sp)      /* arg9  */
sd   x19, 8(sp)      /* arg10 */
jal  ra, callee10
addi sp, sp, 32
```

Callee side loads overflow arguments (offsets depend on ABI-defined layout):

```
/* Callee side (conceptual): load arg9 and arg10 from stack overflow
 ↪  area */
ld   x5,  0(sp)      /* arg9  */
ld   x6,  8(sp)      /* arg10 */
```

Aggregates and mixed types (conceptual rule)

ABIs typically pass small scalars in registers and place larger aggregates (structs, arrays) either in registers if they fit specific rules or by memory (stack or hidden pointer). This booklet focuses on the discipline:

- If you hand-write assembly intended to match C/C++, follow the ABI's exact classification rules for the target platform.
- When in doubt, pass aggregates by pointer explicitly.

Example: explicitly passing a pointer to a struct:

```
/* void g(struct S* p, uint64_t v)
   a0 = p, a1 = v
*/
sd   x11, 0(x10)      /* p->field0 = v (conceptual) */
ret
```

9.2 Return Value Rules

The ABI defines return locations for results:

- Integer and pointer return values are placed in `a0`.
- Some ABIs allow a second return register (`a1`) for multiword returns or paired scalars.
- Large aggregates are typically returned via a hidden pointer provided by the caller (sret style), pointing to caller-allocated storage.

Single integer/pointer return

```
/* int64_t add2(int64_t x, int64_t y) */
add  x10, x10, x11   /* return in a0 */
ret
```

Returning a pair of values (conceptual)

When the ABI supports two-register returns:

```
/* Conceptual: return (sum, diff) in a0 and a1 */
add  x10, x10, x11   /* a0 = sum */
sub  x11, x10, x11   /* a1 = diff (illustration; requires careful
 ↪  ordering) */
ret
```

A safer version that preserves operands:

```
/* a0=x, a1=y; return a0=x+y, a1=x-y */
add  x12, x10, x11   /* t0 = x+y */
sub  x13, x10, x11   /* t1 = x-y */
add  x10, x12, x0    /* a0 = t0 */
add  x11, x13, x0    /* a1 = t1 */
ret
```

Returning large aggregates (conceptual hidden pointer)

A common ABI strategy is caller-allocated return storage passed as a hidden first argument. The callee writes into that memory and returns normally.

```
/* Conceptual:
```

```
   a0 = sret pointer (hidden)
   a1.. = user arguments
   callee writes result to *(a0)
*/
sd   x11, 0(x10)      /* store part of result */
sd   x12, 8(x10)      /* store part of result */
ret
```

9.3 Stack Layout and Alignment

The ABI defines the stack as the primary memory structure for:

- preserving callee-saved registers
- saving `ra` in non-leaf functions
- local variables and spill slots
- outgoing argument overflow area (when needed)

Alignment requirement

A central ABI rule on RV64 systems is that `sp` must be aligned at call boundaries (commonly 16-byte alignment). This is a contract used by compilers and runtime tooling.
Practical rule:

- Adjust `sp` by a multiple of the required alignment.
- Preserve alignment before executing `jal` to another function.

Leaf function: no frame required

A leaf function that makes no calls and uses only caller-saved registers may avoid stack usage entirely:

```
/* Leaf: no calls, no callee-saved usage */
add  x10, x10, x11
xor  x10, x10, x12
ret
```

Non-leaf function: save ra and callee-saved regs

A function that calls another function must preserve its return path by saving `ra` if it will be overwritten by a nested call. If it uses any callee-saved registers, it must save/restore them.

```
/* Non-leaf function with a small aligned frame */
addi sp, sp, -32
sd   x1,  24(sp)     /* save ra */
sd   x8,  16(sp)     /* save callee-saved (example) */

/* body: compute stable local in x8 */
add  x8, x10, x11

jal  ra, helper

add  x10, x8, x0     /* return value in a0 */

ld   x8,  16(sp)
ld   x1,  24(sp)
addi sp, sp, 32
ret
```

Outgoing argument area (conceptual)

If the caller must pass stack arguments (beyond `a0--a7`), it must allocate space and store them before the call, without breaking alignment.

```
/* Caller passes 9th and 10th args on stack (conceptual) */
addi sp, sp, -48     /* keep alignment while reserving outgoing area
 ↪  */
sd   x18, 0(sp)      /* arg9 */
sd   x19, 8(sp)      /* arg10 */
jal  ra, callee10
addi sp, sp, 48
```

Discipline summary

Correct RV64 call boundaries depend on three non-negotiable rules:

- Put arguments where the ABI expects them.
- Return results where the ABI expects them.
- Preserve stack alignment and save/restore required registers.

Violating these rules often does not crash immediately. Instead, it produces silent corruption, incorrect returns, and failures that appear “random” under optimization or when mixing libraries and toolchains.

Chapter 10

Function Prologues and Epilogues

10.1 Register Preservation Rules

A function prologue/epilogue is the ABI-enforced mechanism that preserves program correctness across calls. The core rule is:

- Caller-saved registers may be clobbered by a call; the caller must save anything it needs after the call.
- Callee-saved registers must be preserved by the callee if it uses them.

In RV64 ABIs, registers commonly used as callee-saved include `s0--s11` (architecturally `x8--x9` and `x18--x27`). The return address `ra` is caller-saved (it is overwritten by a nested call), so a non-leaf function must save it if it performs any call.

Caller-side preservation (caller-saved)

If the caller wants to keep a value currently in a caller-saved register (e.g., a temporary) across a call, it must spill it:

```
/* Caller preserves a live temporary across a call */
addi sp, sp, -16
sd   x5, 8(sp)       /* save live value in caller-saved register */
jal  ra, helper
ld   x5, 8(sp)       /* restore */
addi sp, sp, 16
```

Callee-side preservation (callee-saved)

If a callee wants to use a callee-saved register, it must save and restore it:

```
/* Callee uses x8 as a stable local; must preserve it */
addi sp, sp, -16
sd   x8, 8(sp)

/* body */
add  x8, x10, x11

ld   x8, 8(sp)
addi sp, sp, 16
ret
```

Saving ra in non-leaf functions

A non-leaf function that makes a call must preserve its return path. Since `jal` writes `ra`, the function must save `ra` before any nested call:

```
/* Non-leaf: save ra, call helper, return safely */
addi sp, sp, -16
sd   x1, 8(sp)       /* save ra */
```

```
jal  ra, helper

ld   x1, 8(sp)        /* restore ra */
addi sp, sp, 16
ret
```

10.2 Stack Frame Construction

A stack frame is the memory region reserved by a function for:

- saved callee-saved registers
- saved return address (when needed)
- local variables
- spill slots created by register allocation
- optional outgoing argument space (for stack-passed args)

A correct RV64 frame must preserve stack alignment at call boundaries (commonly 16-byte aligned). The practical rule is:

- Subtract a multiple of the required alignment from `sp`.
- Use fixed offsets from `sp` for saved registers and locals.
- Restore `sp` exactly in the epilogue.

Minimal aligned frame saving ra and one saved register

```
/* Frame layout (example):
   sp+24: saved ra
   sp+16: saved x8
   sp+0..15: local/spill (unused in this minimal example)
*/
addi sp, sp, -32
sd   x1,  24(sp)
sd   x8,  16(sp)

/* body */
add  x8, x10, x11    /* stable local in x8 */
jal  ra, helper
add  x10, x8, x0     /* return value in a0 */

/* epilogue */
ld   x8,  16(sp)
ld   x1,  24(sp)
addi sp, sp, 32
ret
```

Local storage and stack-based temporaries

Example: a function that needs two local 64-bit slots and makes a call:

```
/* Reserve space: saved regs + 16 bytes locals */
addi sp, sp, -48
sd   x1,  40(sp)     /* ra */
sd   x8,  32(sp)     /* s0 */
```

```
/* locals at sp+0 and sp+8 */
sd   x10, 0(sp)       /* local0 = a0 */
sd   x11, 8(sp)       /* local1 = a1 */

jal  ra, helper

ld   x5, 0(sp)        /* reload local0 */
ld   x6, 8(sp)        /* reload local1 */
add  x10, x5, x6      /* return local0+local1 */

ld   x8,  32(sp)
ld   x1,  40(sp)
addi sp, sp, 48
ret
```

10.3 Typical Compiler-Generated Patterns

Compilers generate prologues and epilogues according to optimization level, frame size, and which registers are used. However, common patterns are stable because they are driven by ABI constraints.

Leaf function pattern

A leaf function that does not call others and uses only caller-saved registers often has no frame:

```
/* Leaf: no stack, no callee-saved, no calls */
add  x10, x10, x11
ret
```

Non-leaf with frame pointer (conceptual)

Many toolchains optionally use a frame pointer for debugging or when frame addressing is complex. A common convention is to use `s0` (`x8`) as the frame pointer. The frame pointer is typically set to the caller's stack top or a stable location within the frame.

```
/* Typical pattern using x8 as frame pointer (s0) */
addi sp, sp, -32
sd   x1,  24(sp)     /* save ra */
sd   x8,  16(sp)     /* save old fp */
addi x8, sp, 32      /* set fp to prior sp (conceptual stable
 ↪  reference) */

/* body */
jal  ra, helper

ld   x8,  16(sp)     /* restore old fp */
ld   x1,  24(sp)     /* restore ra */
addi sp, sp, 32
ret
```

Spills and reloads

When register pressure is high, the compiler spills temporaries to the stack. These are caller-local decisions but must still preserve alignment and ABI rules.

```
/* Example spill/reload pattern inside a function frame */
addi sp, sp, -32
sd   x1,  24(sp)
```

```
sd   x5,  0(sp)       /* spill temporary */
sd   x6,  8(sp)       /* spill temporary */

jal  ra, helper

ld   x6,  8(sp)       /* reload */
ld   x5,  0(sp)       /* reload */

ld   x1,  24(sp)
addi sp, sp, 32
ret
```

Epilogue as exact inverse

A correct epilogue is the exact inverse of the prologue:

- restore saved registers from fixed offsets
- restore `sp` to its entry value
- return via `ret`

This reversibility is a key property relied on by unwinding, debugging, and exception mechanisms. Any deviation (wrong offsets, misalignment, missing restore) breaks the ABI contract and can corrupt control flow silently.

Chapter 11

C and C++ Interoperability

11.1 Mapping C/C++ Types to the RV64 ABI

Interoperability means that hand-written RV64 assembly must match the ABI expectations produced by a C/C++ compiler. The most important mapping rule is:

- The ABI defines type sizes, alignments, argument locations, and return locations for the target platform.

On most RV64 platforms using an LP64-style model:

- `char` is 8-bit
- `short` is 16-bit
- `int` is 32-bit
- `long` is 64-bit
- `long long` is 64-bit

- pointers are 64-bit

This implies the following practical assembly choices for loads/stores:

- 32-bit `int`: `lw` (signed) or `lwu` (unsigned), store with `sw`
- 64-bit `long`/pointer: `ld`, store with `sd`

Signedness is not optional: choose the correct load

When a 32-bit value is loaded into a 64-bit register, the load instruction determines whether it becomes sign-extended or zero-extended.

```
/* int32_t x = *(int32_t*)p; */
lw   x5, 0(x10)       /* sign-extend */

/* uint32_t u = *(uint32_t*)p; */
lwu  x6, 0(x10)       /* zero-extend */
```

Returning C integers in assembly

For integer/pointer returns, place the result in `a0`:

```
/* int add2(int a, int b)
   a0=a, a1=b (32-bit values in 64-bit regs)
   return in a0 as int
*/
addw x10, x10, x11   /* 32-bit add, sign-extend result */
ret
```

For a 64-bit return type (e.g., `long` or `int64_t`), use 64-bit ops:

```
/* long add2l(long a, long b) */
add  x10, x10, x11
ret
```

Pointer arguments and returns

Pointers are 64-bit. Treat them as such in both registers and memory:

```
/* void store64(uint64_t* p, uint64_t v)
   a0=p, a1=v
*/
sd   x11, 0(x10)
ret
```

11.2 Struct and Aggregate Passing

Struct/aggregate passing is ABI-defined and more complex than scalar passing. A safe interoperability principle is:

- If you must match C/C++ ABI exactly, follow the platform's aggregate classification rules.
- If you control both sides, pass aggregates by pointer explicitly.

Passing a struct by pointer (portable and explicit)

```
/* C:
   struct S { uint64_t a; uint64_t b; };
   void setS(struct S* p, uint64_t x, uint64_t y);
   a0=p, a1=x, a2=y
```

```
*/
sd   x11, 0(x10)      /* p->a = x */
sd   x12, 8(x10)      /* p->b = y */
ret
```

Returning a struct via hidden pointer (conceptual)

Many ABIs return larger aggregates via caller-allocated storage. The caller passes a hidden return pointer; the callee writes the result there and returns normally.

```
/* Conceptual sret pattern:
   a0 = hidden pointer to return storage (struct S*)
   a1.. = user arguments
*/
sd   x11, 0(x10)      /* store first field */
sd   x12, 8(x10)      /* store second field */
ret
```

Small aggregates in registers (conceptual discipline)

Some ABIs may pass or return small aggregates in registers if they fit. When interoperating with compiled C/C++, never guess. Either:

- consult the exact ABI rule set for the target, or
- change the interface to pass by pointer.

A robust engineering practice for toolchains and runtimes is to use explicit pointer-based interfaces at ABI boundaries.

11.3 Variadic Functions

Variadic functions (e.g., `printf`-style) introduce additional ABI requirements because the callee cannot know the number or types of arguments from the function signature alone. The ABI must guarantee that the callee can retrieve variadic arguments consistently.
Two discipline rules dominate variadic interoperability:

- Default argument promotions apply at the C language level.
- The caller must place arguments so that the callee can access them via the ABI-defined mechanism (register save area and/or stack).

Default promotions (C rules that affect ABI reality)

At the call site of a variadic function:

- `float` is promoted to `double`
- integer types smaller than `int` are promoted to `int` (or `unsigned int` depending on representation)

This changes what the callee must expect in registers/stack.

Practical assembly guidance for calling variadic functions

If you hand-write a call to a variadic function, you must:

- place fixed arguments in the normal argument registers
- place variadic arguments following the same register/stack assignment rules
- ensure stack alignment at the call boundary

Conceptual example: calling a variadic function with a format pointer and two integer arguments:

```
/* Call: vfunc(fmt, x, y)
   a0 = fmt pointer
   a1 = x
   a2 = y
*/
jal  ra, vfunc
```

If the argument count exceeds `a0--a7`, overflow arguments are passed on the stack in the outgoing argument area, preserving alignment:

```
/* Conceptual: pass more than 8 arguments to a variadic callee */
addi sp, sp, -48
sd   x18, 0(sp)      /* overflow arg */
sd   x19, 8(sp)      /* overflow arg */
jal  ra, vfunc_many
addi sp, sp, 48
```

Implementing a variadic callee in assembly

Implementing a variadic callee correctly requires the ABI-defined layout for where argument values are stored (register save area and stack). Because this layout is ABI-specific, the safe engineering guidance is:

- Do not implement a variadic callee in hand-written assembly unless you are explicitly following the platform psABI rules for variadic argument access.
- Prefer writing the variadic boundary in C and calling non-variadic helper functions implemented in assembly.

Example pattern: C handles `va_list`, assembly implements a fixed-ABI worker:

```
/* Assembly worker: fixed signature (no varargs) */
add  x10, x10, x11   /* example fixed computation */
ret
```

This approach preserves portability and correctness while still allowing assembly-level optimization where it is safe.

Chapter 12

Practical RV64I Assembly

12.1 ABI-Correct Function Examples

This section provides short, ABI-correct patterns that are safe to link with C/C++ code on RV64 systems. The core discipline is always the same:

- arguments in `a0--a7`
- return in `a0`
- maintain stack alignment at call boundaries
- save/restore `ra` in non-leaf functions
- save/restore any callee-saved registers you modify

Example 1: leaf function (no frame needed)

```
/* int64_t add2(int64_t x, int64_t y)
   a0=x, a1=y, return a0
```

```
*/
.global add2
.type add2, %function
add2:
add  x10, x10, x11
ret
```

Example 2: leaf function with 32-bit semantics

```
/* int add2_i32(int a, int b)
   a0=a, a1=b, return int in a0
*/
.global add2_i32
.type add2_i32, %function
add2_i32:
addw x10, x10, x11   /* 32-bit add, sign-extend */
ret
```

Example 3: non-leaf function saving ra

```
/* int64_t f(int64_t x, int64_t y) { helper(); return x+y; }
   a0=x, a1=y
*/
.global f_call_helper
.type f_call_helper, %function
f_call_helper:
addi sp, sp, -16
sd   x1, 8(sp)       /* save ra */

jal  ra, helper
```

```
add  x10, x10, x11   /* return x+y in a0 */

ld   x1, 8(sp)       /* restore ra */
addi sp, sp, 16
ret
```

Example 4: preserve a value across a call using a callee-saved register

```
/* int64_t g(int64_t x, int64_t y) { t=x+y; helper(); return t; }
   Use x8 as stable local (callee-saved)
*/
.global g_keep_local
.type g_keep_local, %function
g_keep_local:
addi sp, sp, -32
sd   x1, 24(sp)      /* save ra */
sd   x8, 16(sp)      /* save callee-saved */

add  x8, x10, x11    /* t in x8 */
jal  ra, helper
add  x10, x8, x0     /* return t */

ld   x8, 16(sp)
ld   x1, 24(sp)
addi sp, sp, 32
ret
```

12.2 Stack-Based Local Variables

Stack locals are required when:

- a value must survive across calls but registers are insufficient
- you need addressable storage (e.g., for passing by pointer)
- the compiler would normally spill temporaries

The ABI requires aligned stack frames. A safe practice is to reserve space in multiples of the required alignment (commonly 16 bytes).

Example 1: two 64-bit locals

```
/* int64_t sum2_saved(int64_t x, int64_t y) { helper(); return x+y; }
   locals store x and y across the call
*/
.global sum2_saved
.type sum2_saved, %function
sum2_saved:
addi sp, sp, -48
sd   x1, 40(sp)       /* ra */

/* locals */
sd   x10, 0(sp)       /* local0 = x */
sd   x11, 8(sp)       /* local1 = y */

jal  ra, helper

ld   x5, 0(sp)
```

```
ld   x6, 8(sp)
add  x10, x5, x6      /* return */

ld   x1, 40(sp)
addi sp, sp, 48
ret
```

Example 2: addressable local passed by pointer

```
/* void write_then_call(uint64_t v) { tmp=v; helper_ptr(&tmp); }
   a0=v
*/
.global write_then_call
.type write_then_call, %function
write_then_call:
addi sp, sp, -32
sd   x1, 24(sp)       /* ra */

sd   x10, 0(sp)       /* tmp at sp+0 */
add  x10, sp, x0      /* a0 = &tmp */
jal  ra, helper_ptr

ld   x1, 24(sp)
addi sp, sp, 32
ret
```

Example 3: outgoing stack arguments (more than a0–a7)

```
/* Call a function with 10 integer args (conceptual):
   a0-a7 = args 1..8
```

```
   stack = args 9..10
*/
.global call10
.type call10, %function
call10:
addi sp, sp, -48     /* keep alignment while reserving outgoing area
 ↪  */
sd   x18, 0(sp)      /* arg9  */
sd   x19, 8(sp)      /* arg10 */
jal  ra, callee10
addi sp, sp, 48
ret
```

12.3 Common ABI Mistakes and Debugging

ABI bugs often compile and run but fail under optimization or when linked with other objects. The following mistakes are the most common.

Mistake 1: not saving ra in a non-leaf function

```
/* WRONG: ra overwritten by helper call, return is corrupted */
bad_no_ra_save:
jal  ra, helper
ret
```

Correct:

```
/* Correct: save/restore ra */
good_ra_save:
addi sp, sp, -16
```

```
sd   x1, 8(sp)
jal  ra, helper
ld   x1, 8(sp)
addi sp, sp, 16
ret
```

Mistake 2: clobbering callee-saved registers

```
/* WRONG: modifies x8 (callee-saved) without preserving it */
bad_clobber_s0:
add  x8, x10, x11
ret
```

Correct:

```
/* Correct: preserve callee-saved x8 */
good_preserve_s0:
addi sp, sp, -16
sd   x8, 8(sp)
add  x8, x10, x11
ld   x8, 8(sp)
addi sp, sp, 16
ret
```

Mistake 3: breaking stack alignment at call boundaries

```
/* WRONG: frame size not aligned to ABI requirement (example) */
bad_align:
addi sp, sp, -24
jal  ra, helper
```

```
addi sp, sp, 24
ret
```

Correct: allocate a multiple of the required alignment:

```
/* Correct: aligned allocation */
good_align:
addi sp, sp, -32
jal  ra, helper
addi sp, sp, 32
ret
```

Mistake 4: wrong load extension for 32-bit values

```
/* WRONG for uint32_t: sign-extends and changes meaning for high bit
 ↪  set */
lw   x5, 0(x10)
```

Correct for unsigned 32-bit:

```
/* Correct for uint32_t: zero-extend */
lwu  x5, 0(x10)
```

Mistake 5: truncating pointers

```
/* WRONG: reads only 32 bits of a 64-bit pointer */
lw   x5, 0(x10)
```

Correct:

```
/* Correct: pointers are 64-bit */
ld   x5, 0(x10)
```

Debugging approach

When debugging ABI issues, focus on invariants at call boundaries:

- Is `sp` aligned before every `jal`?
- Is `ra` preserved in every non-leaf function?
- Are all modified callee-saved registers restored?
- Are arguments in `a0--a7` and overflow args placed correctly?
- Are loads using correct sign/zero extension for C/C++ types?

A minimal sanity probe for call boundaries is to save and restore only what is required, then reduce the function until the failure disappears. ABI bugs are usually deterministic: once you identify the violated invariant, the fix is mechanical.

Appendices

Appendix A — RV64I vs RV32I Summary

Register Width Comparison

RV32I and RV64I share the same architectural structure: 32 integer registers (`x0--x31`), load/store semantics, and fixed 32-bit instruction encoding for the base ISA. The fundamental difference is XLEN.

- RV32I: XLEN = 32. Integer registers hold 32-bit values.
- RV64I: XLEN = 64. Integer registers hold 64-bit values.

This impacts:

- the natural width of arithmetic operations
- pointer size and address computations
- the set of available instructions (RV64 adds word-ops to preserve 32-bit intent)

Example: same computation, different natural width:

```
/* RV32I-style intent: 32-bit add */
add  x5, x6, x7
```

```
/* RV64I: full 64-bit add (default) */
add  x5, x6, x7

/* RV64I: explicit 32-bit add with sign-extension */
addw x5, x6, x7
```

In RV64I, `add` computes in 64-bit. If your algorithm is defined in 32-bit arithmetic (modulo 2^{32} or language-level `int`), you must use word-ops (`addw/addiw`, etc.) to preserve semantics.

Addressing Differences

Both RV32I and RV64I use base+offset addressing for memory operations, but the address width differs:

- RV32I addresses are 32-bit (pointers are 32-bit).
- RV64I addresses are 64-bit (pointers are 64-bit).

The most important practical consequences are:

- Pointers must never be truncated in RV64I.
- Pointer-sized objects in memory must be accessed with `ld/sd`.
- 32-bit loads into 64-bit registers require correct sign/zero extension.

Correct pointer load in RV64I:

```
/* x10 points to a stored pointer (uint64_t) */
ld   x5, 0(x10)       /* x5 = *(uint64_t*)x10 (64-bit pointer) */
```

Incorrect (typical bug):

```
/* WRONG: reads only 32 bits of a 64-bit pointer */
lw   x5, 0(x10)      /* truncates then sign-extends */
```

Index scaling becomes more visible on RV64 systems because element size is often pointer-sized (8 bytes):

```
/* Load a[i] where a is uint64_t*, i in x11, base in x10 */
slli x12, x11, 3     /* i * 8 */
add  x12, x10, x12
ld   x5, 0(x12)
```

ABI-Level Impact

The ISA defines execution; the ABI defines interoperability. Moving from RV32 to RV64 changes the ABI in ways that affect every compiled boundary:

- Register argument passing still uses `a0--a7`, but arguments and pointers are now 64-bit wide.
- Stack slots for spilled values, locals, and overflow arguments become pointer-sized where appropriate (8-byte).
- Stack alignment requirements are stricter and must be preserved at call boundaries (commonly 16-byte aligned).
- Object layout changes under 64-bit data models (e.g., LP64): `long` and pointers are 64-bit; `int` remains 32-bit.

Example: ABI-correct non-leaf function in RV64I that preserves `ra` and a callee-saved register, maintains alignment, and returns through `a0`:

```
/* int64_t g(int64_t x, int64_t y) { t=x+y; helper(); return t; } */
addi sp, sp, -32
sd   x1, 24(sp)       /* save ra */
sd   x8, 16(sp)       /* save callee-saved */

add  x8, x10, x11     /* t in x8 */
jal  ra, helper
add  x10, x8, x0      /* return t in a0 */

ld   x8, 16(sp)
ld   x1, 24(sp)
addi sp, sp, 32
ret
```

A frequent RV64 migration error is preserving old RV32 assumptions:

- using `lw/sw` where `ld/sd` is required for pointers or `long`
- using signed loads (`lw`) for unsigned 32-bit values instead of `lwu`
- breaking stack alignment in hand-written assembly

Practical summary:

- RV64I keeps the RV32I philosophy and instruction structure.
- The widened machine state makes pointer and ABI correctness mandatory.
- Correct RV64 code is explicit about width, extension, and call boundaries.

Appendix B — RV64 ABI Register Usage

Caller-Saved Registers

Caller-saved means: a function call may clobber these registers. If the caller needs a value after the call, the caller must save it (typically on the stack) before calling and restore it after. In common RV64 ABIs, the following are treated as caller-saved categories:

- `a0--a7` (x10–x17): argument/return registers (also caller-saved)
- `t0--t6` (x5–x7, x28–x31): temporaries
- `ra` (x1): return address (overwritten by nested calls)

Example: preserve a live caller-owned value across a call:

```
/* Caller wants to preserve x5 across helper() */
addi sp, sp, -16
sd   x5, 8(sp)       /* save caller-saved register */
jal  ra, helper
ld   x5, 8(sp)       /* restore */
addi sp, sp, 16
```

Example: preserve argument registers across a call when needed:

```
/* Caller needs original a0/a1 after calling helper */
addi sp, sp, -32
sd   x10, 16(sp)     /* save a0 */
sd   x11, 8(sp)      /* save a1 */
jal  ra, helper
ld   x11, 8(sp)      /* restore a1 */
ld   x10, 16(sp)     /* restore a0 */
addi sp, sp, 32
```

Callee-Saved Registers

Callee-saved means: if a function uses these registers, it must preserve and restore them so the caller observes the same values after the call.
In common RV64 ABIs, the following are callee-saved categories:

- `s0--s11` (x8–x9, x18–x27): saved registers
- `sp` (x2): stack pointer must be restored to entry value

Minimal example: callee uses x8 and preserves it:

```
/* Callee modifies x8, must preserve it */
addi sp, sp, -16
sd   x8, 8(sp)

add  x8, x10, x11    /* use x8 as a stable local */

ld   x8, 8(sp)
addi sp, sp, 16
ret
```

Non-leaf example: callee uses x8 and calls another function, preserving both x8 and ra:

```
/* Callee uses x8 and makes a call */
addi sp, sp, -32
sd   x1, 24(sp)      /* save ra */
sd   x8, 16(sp)      /* save callee-saved */

add  x8, x10, x11    /* stable local in x8 */
jal  ra, helper
```

```
add  x10, x8, x0      /* return value in a0 */

ld   x8, 16(sp)
ld   x1, 24(sp)
addi sp, sp, 32
ret
```

Special-Purpose Registers

Special-purpose registers have ABI-defined roles that must be respected for correct execution and interoperability.

- `x0`: constant zero (writes ignored)
- `ra` (x1): return address (caller-saved; save in non-leaf functions)
- `sp` (x2): stack pointer (must maintain ABI alignment at call boundaries)
- `gp` (x3): global pointer (platform-defined; do not clobber)
- `tp` (x4): thread pointer (platform-defined; do not clobber)

Return address behavior:

```
/* jal writes ra; nested calls overwrite ra */
jal  ra, callee
ret
```

Stack pointer alignment discipline (common requirement: keep sp aligned at calls):

```
/* Correct: allocate aligned frame before a call */
addi sp, sp, -32
jal  ra, helper
addi sp, sp, 32
```

Zero register idioms:

```
/* Clear and move using x0 */
add  x5, x0, x0      /* x5 = 0 */
add  x6, x5, x0      /* x6 = x5 */
```

Practical rules for ABI-safe code:

- Treat `gp` and `tp` as reserved unless you fully control the platform ABI.
- Save `ra` in every non-leaf function.
- Preserve every callee-saved register you modify.
- Assume caller-saved registers are destroyed by any call.
- Maintain stack alignment at every call boundary.

Appendix C — Common 64-Bit Addressing Errors

Pointer Truncation

Pointer truncation is the most destructive RV64 bug: treating a 64-bit pointer as a 32-bit value destroys the high address bits. This can silently redirect memory accesses, cause faults, or corrupt unrelated memory.

Wrong pattern: loading/storing pointers with 32-bit instructions

```
/* WRONG: reads only low 32 bits of a 64-bit pointer */
lw   x5, 0(x10)      /* truncates pointer then sign-extends */

/* WRONG: writes only low 32 bits, corrupting stored pointer */
sw   x5, 8(x10)
```

Correct pattern: pointers are 64-bit objects

```
/* Correct: load/store pointer-sized values with ld/sd */
ld   x5, 0(x10)       /* x5 = *(uint64_t*)x10 */
sd   x5, 8(x10)       /* *(uint64_t*)(x10+8) = x5 */
```

Common real-world trigger: using `lw` for address tables

```
/* WRONG: jump table entry is 64-bit pointer, but loaded as 32-bit */
lw   x5, 0(x10)
jalr x0, 0(x5)        /* jumps to garbage/non-canonical address */
```

Correct:

```
/* Correct: load 64-bit target address */
ld   x5, 0(x10)
jalr x0, 0(x5)
```

Practical rule:

- Any value that represents an address must be loaded/stored as 64-bit.
- Never cast or mask pointers to 32-bit unless the ABI explicitly requires it.

Misaligned Stack Frames

On RV64 ABIs, the stack pointer must satisfy an alignment constraint at call boundaries (commonly 16-byte aligned). Misalignment breaks assumptions used by compilers, spill code, and runtime tooling, and it can cause failures that appear only under optimization or when linking against libraries.

Wrong pattern: allocating a non-aligned frame size

```
/* WRONG: -24 breaks common 16-byte alignment requirement */
bad_frame:
addi sp, sp, -24
jal  ra, helper
addi sp, sp, 24
ret
```

Correct pattern: allocate a multiple of alignment

```
/* Correct: -32 preserves 16-byte alignment */
good_frame:
addi sp, sp, -32
jal  ra, helper
addi sp, sp, 32
ret
```

Wrong pattern: saving registers at miscomputed offsets

Even if the allocation is aligned, incorrect offsets corrupt the restore step:

```
/* WRONG: save at one offset, restore from another */
addi sp, sp, -32
sd   x1, 24(sp)
jal  ra, helper
ld   x1, 16(sp)      /* wrong offset: restores garbage */
addi sp, sp, 32
ret
```

Correct:

```
/* Correct: exact inverse offsets */
addi sp, sp, -32
```

```
sd   x1, 24(sp)
jal  ra, helper
ld   x1, 24(sp)
addi sp, sp, 32
ret
```

Practical rule:

- Prologue and epilogue must be exact inverses.
- Keep frame size aligned and offsets consistent.
- Save `ra` in every non-leaf function.

Incorrect Sign Extension

RV64 makes sign/zero extension unavoidable: loading a 32-bit value into a 64-bit register must choose an extension rule. Using the wrong rule changes program meaning, especially when the high bit is set.

Classic bug: using `lw` for `uint32_t`

```
/* WRONG for uint32_t: lw sign-extends */
lw   x5, 0(x10)      /* 0x80000000 becomes 0xFFFFFFFF80000000 */
```

Correct:

```
/* Correct for uint32_t: lwu zero-extends */
lwu  x5, 0(x10)      /* 0x80000000 becomes 0x0000000080000000 */
```

Bug in comparisons: signed vs unsigned branches

If you loaded an unsigned quantity but branch using signed comparison, you can introduce bounds-check errors:

```
/* x11=index (uint32), x12=size (uint32), both zero-extended via lwu
 ↪  */
/* WRONG: signed comparison */
blt  x11, x12, in_range
```

Correct for unsigned bounds:

```
/* Correct: unsigned comparison */
bltu x11, x12, in_range
```

Bug in arithmetic width: losing intended 32-bit semantics

If an algorithm is defined in 32-bit arithmetic, using 64-bit operations can silently change overflow behavior.

```
/* WRONG for 32-bit counter semantics */
lwu  x5, 0(x10)      /* uint32_t */
addi x5, x5, 1       /* 64-bit add changes modulo behavior */
sw   x5, 0(x10)
```

Correct:

```
/* Correct: 32-bit add with sign-extended result */
lwu  x5, 0(x10)
addiw x5, x5, 1
sw   x5, 0(x10)
```

Practical rule:

- Use `lw` for signed 32-bit values, `lwu` for unsigned 32-bit values.
- Use `addw/addiw` when the algorithm is defined in 32-bit arithmetic.
- Use unsigned branches (`bltu/bgeu`) for unsigned range checks.

Appendix D — Cross-References

Related Booklets in This Series

This booklet focuses on RV64I addressing and ABI rules. It is designed to fit into a structured progression where each booklet builds a specific layer of understanding. The following booklets are directly related at the conceptual and practical level:

- **CPU Programming Series — Booklet 12: AArch64 Core Architecture**
  Provides architectural grounding for 64-bit register files, addressing, and privilege separation concepts that parallel RV64 ideas.

- **CPU Programming Series — Booklet 13: AArch64 Calling Convention**
  Establishes ABI reasoning, stack discipline, and C/C++ interoperability concepts that reappear in RV64 under different concrete rules.

- **CPU Programming Series — Booklet 14: AArch64 Exceptions & Syscalls**
  Complements this booklet by showing how ABI rules extend across privilege boundaries and system interfaces.

- **CPU Programming Series — Booklet 15: RISC-V RV32I Assembly**
  Introduces the RV32I base ISA, register model, and ABI fundamentals. Booklet 16 builds directly on this foundation by widening XLEN and ABI rules.

From the RISC-V perspective specifically, this booklet should be read as the 64-bit continuation of RV32I rather than as a separate architecture. The core instruction philosophy remains unchanged; width, addressing, and ABI semantics are what evolve.

Suggested Reading Order

For readers aiming at a clean and cumulative understanding of RV64 systems programming, the following reading order is recommended:

- **Step 1 — RV32I Foundations**
  Read the RV32I booklet to establish the base ISA model: registers, load/store discipline, branches, and minimal ABI concepts.
- **Step 2 — RV64I Architecture and Addressing**
  Read Chapters 1–6 of this booklet to understand:
  - XLEN expansion from 32 to 64
  - pointer size changes
  - canonical addressing expectations
  - load/store extension rules
- **Step 3 — RV64 ABI and Calling Convention**
  Read Chapters 7–10 of this booklet to master:
  - 64-bit arithmetic and control flow
  - ABI register classification
  - argument passing and return rules
  - stack frames, prologues, and epilogues
- **Step 4 — Language Interoperability**
  Read Chapter 11 to connect RV64 assembly with C/C++:
  - type mapping under LP64
  - struct passing strategies

  - variadic function constraints

- **Step 5 — Practical Patterns and Pitfalls**
  Read Chapter 12 and Appendices A–C to internalize:

  - ABI-correct hand-written assembly
  - stack-local design
  - common 64-bit migration errors

Conceptual Progression Across the Series

The broader intent of the series is cumulative:

- RV32I teaches *how instructions work*.
- RV64I teaches *how programs and ABIs scale*.
- Later booklets connect these rules to:

  - operating systems
  - toolchains
  - system calls and privilege transitions

By the end of this booklet, the reader should be able to:

- read compiler-generated RV64 assembly with confidence
- write ABI-correct hand assembly callable from C/C++
- diagnose subtle 64-bit bugs related to pointers, alignment, and extension
- reason about RV64 code as a system-level contract, not just instructions

This appendix exists to emphasize that RV64I mastery is not isolated knowledge, but a precise continuation of architectural, ABI, and systems principles introduced earlier in the series.

References

RISC-V RV64I Architectural Specifications

This booklet is grounded in the official RISC-V architectural definition of the base integer ISA and its 64-bit instantiation. The RV64I specification defines:

- XLEN = 64 register width and its implications on arithmetic and addressing
- the uniform 32-register integer file (`x0--x31`)
- load/store architecture and base+offset addressing
- instruction semantics shared with RV32I and extended for 64-bit operation
- word instructions (`W`-suffix) introduced to preserve 32-bit intent

Architectural rules referenced throughout this booklet include:

- modulo 2^{64} arithmetic semantics
- absence of condition flags
- explicit signed vs unsigned instruction forms
- defined behavior of shifts, comparisons, and jumps under RV64

All examples and explanations assume strict adherence to the architectural rules defined for RV64I, without relying on implementation-specific behavior.

RISC-V psABI Documentation

The RISC-V psABI (Processor-Specific Application Binary Interface) defines the binary contract that governs interoperability between separately compiled objects on RV64 systems. Key ABI topics referenced in this booklet include:

- integer and pointer argument passing rules
- return value placement
- caller-saved and callee-saved register classification
- stack growth direction, frame layout, and alignment constraints
- rules for struct, aggregate, and variadic function handling
- preservation of program state across call boundaries

All calling convention examples, stack layouts, and prologue/epilogue patterns in this booklet are consistent with psABI-defined behavior and are suitable for interoperation with standard RISC-V toolchains.

Compiler and Toolchain Behavioral References

This booklet reflects behavior observed in modern RISC-V compiler toolchains that implement the RV64 psABI correctly. The following aspects are reflected consistently across compilers:

- generation of aligned stack frames for non-leaf functions
- use of `a0--a7` for integer and pointer arguments
- spilling of temporaries and locals to stack under register pressure
- use of `addw/addiw` and related word instructions for 32-bit C semantics on RV64
- preservation of callee-saved registers and `ra` as dictated by ABI

Compiler-generated patterns shown in this booklet are representative of optimized and non-optimized builds, emphasizing ABI invariants rather than compiler-specific code generation strategies.
The intent is not to mirror a single compiler, but to document the stable, ABI-driven behavior that all conforming toolchains must follow.

64-Bit Systems Programming Foundations

The conceptual foundation of this booklet aligns with established principles of 64-bit systems programming across architectures. These principles include:

- strict separation between ISA rules and ABI contracts
- pointer-width correctness as a first-order invariant
- explicit handling of sign extension and zero extension
- disciplined stack usage and alignment
- avoidance of implicit assumptions inherited from 32-bit environments

The discussion of common errors (pointer truncation, misalignment, incorrect extension) reflects well-known failure modes encountered when transitioning from 32-bit to 64-bit systems programming.
This booklet treats RV64I not as a collection of instructions, but as a coherent execution and interoperability model suitable for:

- low-level runtime development
- operating system components
- language toolchains
- performance-critical system libraries

Together, these references define the authoritative technical context for all material presented in *My CPU Programming Series Booklet 16: RISC-V RV64I — 64-Bit Addressing & ABI Rules*.