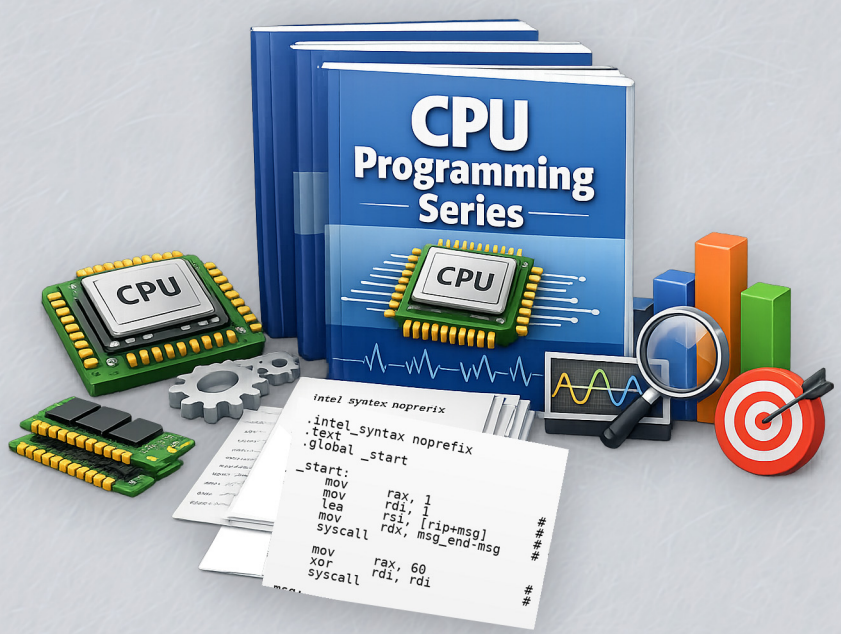
https://simplifycpp.org

CPU Programming Series

AArch64 Core Architecture

Registers, PSTATE, and Addressing



12

Prepared by Ayman Alheraki

CPU Programming Series

AArch64 Core Architecture

Registers, PSTATE, and Addressing

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Purpose of This Booklet

This booklet provides a precise and architecture-focused explanation of the AArch64 core design, concentrating on three foundational pillars: the general-purpose register file, the PSTATE processor state, and the AArch64 addressing model. Its purpose is to build a correct mental model of how 64-bit ARM processors expose state and memory access at the ISA level, without relying on operating systems, ABIs, or compiler conventions. By the end of this booklet, the reader should be able to read, reason about, and manually write AArch64 assembly code that uses registers, flags, and addressing modes correctly and predictably.

Position in the CPU Programming Series

This booklet belongs to the architecture-specific phase of the CPU Programming Series and represents the first dedicated AArch64 volume. It assumes mastery of the shared foundations introduced earlier in the series, including instruction execution, binary data representation, flags, and basic stack concepts. This volume serves as a mandatory prerequisite for subsequent AArch64-focused booklets covering stack discipline, calling conventions, exception levels, privilege transitions, atomics, and memory ordering.

Prerequisites and Assumed Knowledge

The reader is expected to already understand:

- How a CPU fetches, decodes, executes, and retires instructions
- Signed and unsigned integer representation and flag semantics
- The conceptual role of registers, memory, and load/store architectures
- Basic assembly reading skills and instruction sequencing

No prior knowledge of ARM-specific syntax is required, but familiarity with at least one other ISA (such as x86 or ARM 32-bit) is strongly recommended.

What This Booklet Covers — and What It Explicitly Does Not

This booklet covers:

- AArch64 general-purpose registers and width rules
- Special registers such as SP, LR, and PC
- PSTATE flags and execution control bits
- Address calculation and AArch64 addressing modes
- Load and store behavior at the ISA level

This booklet explicitly does **not** cover:

- Any ABI or calling convention rules

- Operating system interfaces or system calls
- Exception levels, privilege transitions, or MMU internals
- Cache hierarchy, coherence protocols, or memory ordering
- Compiler optimizations or language-level semantics

These topics are intentionally deferred to later, dedicated volumes.

Reading Discipline for AArch64

AArch64 enforces strict architectural rules that differ significantly from legacy ISAs. Readers are strongly advised to:

- Treat register width rules as mandatory, not advisory
- Never assume implicit behavior beyond what the ISA guarantees
- Distinguish clearly between architectural behavior and ABI conventions
- Validate every addressing mode against its documented constraints

All examples in this booklet are written to emphasize correctness, predictability, and architectural intent rather than brevity or compiler mimicry. This discipline is essential for reliable low-level programming, debugging, and performance analysis on modern ARM systems.

Chapter 1

AArch64 Architecture Overview

1.1 AArch64 vs AArch32: Architectural Shift

AArch64 is a distinct 64-bit execution environment with its own architectural state, instruction set rules, and system model. While both AArch32 and AArch64 follow a load/store design, the shift to AArch64 introduces a cleaner core programming model that is optimized for 64-bit computation, modern address spaces, and uniform register usage.

Key architectural shifts

- **Register model modernization:** AArch64 exposes a larger, uniform general-purpose register file with consistent 64-bit semantics; 32-bit access is a defined view of the same registers.
- **State model change:** AArch64 uses PSTATE as the architectural processor state for flags and controls; legacy AArch32 state constructs are not directly preserved as-is.
- **Instruction set redesign:** AArch64 is not a superset of AArch32; it uses a distinct fixed-

length encoding space and instruction forms.

- **Addressing and pointer reality:** 64-bit pointers and addressing rules are first-class in AArch64, shaping load/store and address calculation patterns.

Example 1: 32-bit writes have architectural meaning in AArch64

In AArch64, `Wn` is the low 32-bit view of `Xn`. A fundamental architectural rule is that writing `Wn` zero-extends into `Xn`. This is not "compiler behavior"; it is guaranteed by the ISA.

```
.text
.global demo_w_to_x
demo_w_to_x:
    mov     x0, -1
    mov     w0, 1           /* writing W0 zeros the upper 32 bits of
     ↪  X0 */
    ret
```

Example 2: Arithmetic stays in registers; memory access is explicit

AArch64 preserves strict load/store discipline: arithmetic instructions operate on registers, while loads and stores perform memory access.

```
.text
.global demo_load_store
demo_load_store:
    ldr     x1, [x0]
    add     x1, x1, #8
    str     x1, [x0]
    ret
```

1.2 64-bit Design Goals and Constraints

AArch64 is designed to provide a predictable, scalable, and efficient 64-bit architecture. Its design emphasizes a uniform register model, regular instruction decoding, and explicit memory operations. These goals reduce ambiguity and make the ISA suitable for both low-level systems work and high-performance workloads.

Design goals

- **Uniformity:** consistent register widths, consistent operand rules, and reduced special-case behavior.
- **Scalability:** natural support for large address spaces and modern system software structures.
- **Predictability:** fixed instruction length and clearly defined interactions between register views and flags.
- **Efficiency:** addressing modes and instruction forms that reduce instruction count without hiding architectural behavior.

Constraints you must treat as hard rules

- **Fixed-length instructions:** all instructions are 32 bits, shaping decode regularity and alignment assumptions.
- **Load/store separation:** memory operands are restricted to load/store instructions; general arithmetic does not operate directly on memory.
- **Immediate encoding limits:** many constants cannot fit in a single instruction and require canonical construction patterns.

Example: Efficient address usage without mixing memory and ALU

Addressing modes can reduce instruction count while preserving strict load/store discipline.

```
.text
.global sum_two_u32
sum_two_u32:
    ldr     w1, [x0]        /* load element 0 */
    ldr     w2, [x0, #4]    /* load element 1 */
    add     w0, w1, w2      /* sum in registers */
    ret
```

Example: Canonical constant construction

AArch64 commonly constructs a 64-bit constant using `MOVZ` and `MOVK` with shifts.

```
.text
.global build_u64
build_u64:
    movz    x0, #0x1122, lsl #48
    movk    x0, #0x3344, lsl #32
    movk    x0, #0x5566, lsl #16
    movk    x0, #0x7788, lsl #0
    ret
```

1.3 Execution State: AArch64 Fundamentals

A CPU may support multiple execution states. This booklet focuses on the AArch64 execution state as the environment in which AArch64 instructions execute and architectural state is defined.

Core architectural state (AArch64 view)

- **General-purpose registers:** `X0--X30` (64-bit) and `W0--W30` (32-bit views).
- **SP:** the stack pointer, with architectural usage rules and instruction constraints.
- **LR (X30):** link register used by call/return patterns.
- **PC:** program counter, updated by control-flow instructions (not a general-purpose register).
- **PSTATE:** processor state containing condition flags and control bits (detailed in later chapters).

Example: PC is controlled via branches, not arithmetic

Control flow is expressed through branch instructions; the architectural PC update is implicit in branch semantics.

```
.text
.global branch_example
branch_example:
    cmp     w0, #0
    b.eq    is_zero
    mov     w0, #1
```

```
    ret
is_zero:
    mov     w0, #0
    ret
```

Example: Flag-setting must be intentional

Conditional branches read condition flags in PSTATE. The programmer must track which instructions set flags.

```
.text
.global flag_discipline
flag_discipline:
    subs    x1, x0, #1      /* subtract and set flags */
    b.mi    was_negative
    mov     w0, #0
    ret
was_negative:
    mov     w0, #1
    ret
```

1.4 Instruction Length and Encoding Model

AArch64 uses a fixed 32-bit instruction length. This creates a regular decode model and ensures instruction fetch alignment naturally matches 4-byte boundaries. Immediate encoding is necessarily constrained by the 32-bit instruction size, so the ISA defines canonical multi-instruction patterns for tasks such as building 64-bit constants or forming full addresses.

Example: PC-relative addressing pattern for position-independent code

A common architectural pattern for forming the address of data near the instruction stream uses `ADRP` with a low 12-bit add. This is an ISA-level address formation mechanism.

```
.text
.global load_data_pc_relative
load_data_pc_relative:
    adrp    x0, my_data
    add     x0, x0, :lo12:my_data
    ldr     w0, [x0]
    ret

.data
.align 2
my_data:
    .word   123
```

Example: Branch targets are instruction-aligned

Because instructions are 4-byte units, control-flow targets are naturally expressed on instruction boundaries.

```
.text
.global simple_loop
simple_loop:
    mov     w1, #3
loop:
    subs    w1, w1, #1
    b.ne    loop
    ret
```

1.5 Conceptual Execution Pipeline (ISA View)

This booklet uses an ISA-level execution model to reason about correctness without relying on implementation-specific microarchitecture. At the ISA view, instruction behavior can be understood as a sequence of conceptual stages:

- **Fetch:** obtain the 32-bit instruction at the address in PC.
- **Decode:** interpret opcode, register operands, and immediates.
- **Execute:** perform ALU work, compute effective addresses, and decide branches.
- **Memory:** for loads/stores, access memory using the effective address.
- **Writeback:** commit results to destination registers and/or architectural flags/state.

Two discipline rules follow directly:

1. **Separate effective address calculation from memory effects.** Addressing mode selection defines the effective address; memory access is a distinct architectural action.
2. **Treat flags as explicit state.** Conditional branches depend on the most recent flag-setting instruction relevant to the condition.

Example: Effective address calculation vs memory effect

The addressing mode defines the effective address; the load/store performs the memory access at that address.

```
.text
.global pipeline_view_load_store
pipeline_view_load_store:
```

```
    ldr     x1, [x0, #16]   /* EA = X0 + 16; then memory read into X1
    ↪  */
    add     x1, x1, #3      /* ALU work in registers */
    str     x1, [x0, #16]   /* EA = X0 + 16; then memory write from
    ↪  X1 */
    ret
```

Example: Flags persist unless overwritten by a flag-setting instruction

A non-flag-setting instruction between a flag-setting instruction and a conditional branch does not change the flags. The branch uses the previously established flags.

```
.text
.global flag_dependency_is_explicit
flag_dependency_is_explicit:
    subs    w1, w0, #10     /* sets flags from (W0 - 10) */
    add     w2, w2, #1      /* does NOT set flags */
    b.lt    less_than_10    /* uses flags from SUBS */
    mov     w0, #0
    ret
less_than_10:
    mov     w0, #1
    ret
```

Practical reading method used throughout this series

1. Identify register widths (`W` vs `X`) and apply architectural width rules.

2. Mark flag-defining instructions (`CMP`, `TST`, `ADDS`, `SUBS`, etc.).

3. For each load/store, write the effective address: base + offset/register + addressing mode.

4. Treat branches as architectural PC updates, not arithmetic expressions.

Chapter 2

General-Purpose Register File

2.1 X Registers (X0–X30): Structure and Width

AArch64 defines 31 general-purpose integer registers `X0`–`X30`. Each `Xn` is a full **64-bit** architectural register used for:

- integer arithmetic and logic
- pointer and address calculations
- loop counters and indices
- passing values between instructions and basic control-flow patterns

There is no separate architectural bank for "32-bit registers". Instead, `Wn` is a defined view of the same physical/architectural register (covered next).

Example: 64-bit arithmetic uses X registers

```
.text
```

```
.global add_u64
add_u64:
    add     x0, x0, x1      /* X0 = X0 + X1 (64-bit) */
    ret
```

Example: addresses and pointers are 64-bit in core programming

```
.text
.global add_ptr_offset
add_ptr_offset:
    add     x0, x0, #32     /* X0 = X0 + 32 (pointer arithmetic) */
    ret
```

2.2 W Registers and Zero Extension Rules

`W0`–`W30` are the **low 32-bit views** of `X0`–`X30`. The critical AArch64 rule is:

> **Any write to a W register zeros the upper 32 bits of the corresponding X register.**

This zero-extension behavior is an architectural guarantee. It is central to correctness when mixing 32-bit and 64-bit operations.

Example: write to W0 zeroes the upper half of X0

```
.text
.global w_write_zero_ext
w_write_zero_ext:
    mov     x0, -1          /* X0 = 0xFFFF_FFFF_FFFF_FFFF */
    mov     w0, #1          /* X0 becomes 0x0000_0000_0000_0001 */
    ret
```

Example: 32-bit arithmetic produces a zero-extended 64-bit result

```
.text
.global add_u32_zero_ext
add_u32_zero_ext:
    add     w0, w0, w1      /* 32-bit add; result is in W0 and
     ↪  zero-extends into X0 */
    ret
```

2.3 Register Aliasing Rules (Xn vs Wn)

For each register number `n` in `0..30`:

- `Xn` refers to the full 64-bit register.
- `Wn` refers to bits `[31:0]` of the same register.
- Writing `Wn` clears bits `[63:32]` of `Xn`.

This aliasing is not optional and not toolchain-dependent; it is a core part of the AArch64 architecture model.

Example: mixing X and W views safely

```
.text
.global mix_x_w_views
mix_x_w_views:
    mov     x2, #0
    mov     w2, #0xFFFF_FFFF    /* X2 becomes 0x0000_0000_FFFF_FFFF */
    lsr     x2, x2, #16         /* shift as 64-bit */
    ret
```

Example: why “preserving upper bits” fails with W writes

A common mistake is assuming W operations preserve the upper 32 bits. They do not.

```
.text
.global misconception_preserve_upper
misconception_preserve_upper:
    mov     x0, #0x1122334455667788
```

```
    add     w0, w0, #1        /* upper 32 bits are cleared here */
    ret
```

2.4 Architectural Zero Register (XZR / WZR)

AArch64 provides a special architectural register:

- `XZR` acts as a **64-bit constant zero** when used as a source.
- `WZR` acts as a **32-bit constant zero** when used as a source.
- When used as a destination, writes are **discarded** (the result is not stored anywhere).

This is extremely useful for explicit zeroing, comparisons, and intentionally discarding results without consuming a general-purpose register.

Example: produce zero without immediate encoding concerns

```
.text
.global zero_with_zr
zero_with_zr:
    add     x0, xzr, xzr      /* X0 = 0 */
    ret
```

Example: compare against zero without materializing a zero register

```
.text
.global cmp_against_zero
cmp_against_zero:
    cmp     x0, xzr          /* set flags based on X0 - 0 */
    cset    w0, eq           /* W0 = 1 if equal else 0 */
    ret
```

Example: discard a computed value intentionally

```
.text
.global discard_result
discard_result:
    add     xzr, x0, x1     /* compute but discard */
    ret
```

2.5 Register Access and Instruction Constraints

Although `X0--X30` are general-purpose, AArch64 imposes architectural constraints:

- **Width must match the instruction form:** instructions come in 32-bit and 64-bit forms; using `W` vs `X` selects the width.
- **SP is not a general-purpose X register:** `SP` is encoded and restricted in many instruction forms.
- **Some encodings accept XZR but not SP (and vice versa):** a register name may share encoding space but still be constrained by the instruction class.

Example: width selection is part of the instruction

```
.text
.global width_selection
width_selection:
    add     w0, w0, #1      /* 32-bit add; zero-extends into X0 */
    add     x1, x1, #1      /* 64-bit add; preserves full 64-bit
     ↪  state */
    ret
```

Example: SP usage is constrained to specific instruction forms

The stack pointer participates in address calculations for loads/stores and stack adjustments, but it is not interchangeable with `Xn` in all arithmetic encodings.

```
.text
.global sp_stack_adjust
sp_stack_adjust:
```

```
    sub     sp, sp, #16     /* allocate 16 bytes on stack */
    add     sp, sp, #16     /* deallocate */
    ret
```

Example: register moves often use canonical patterns

Moving values is explicit and width-specific.

```
.text
.global move_examples
move_examples:
    mov     x0, x1          /* 64-bit move */
    mov     w2, w3          /* 32-bit move (zero-extends into X2) */
    ret
```

2.6 Common Misconceptions About Register Width

Misconception 1: "W and X are separate registers"

False. They are views of the same architectural register number. Writing `Wn` modifies `Xn` by clearing the upper 32 bits.

Misconception 2: "32-bit instructions preserve upper 32 bits"

False. In AArch64, 32-bit writes **zero** the upper bits by definition.

Misconception 3: "Using W is just an optimization"

Not necessarily. Using `W` selects a different architectural operation width and may change results (especially shifts, sign extension, overflow behavior, and pointer computations).

Example: a real bug pattern when mixing pointer math with W registers

```
.text
.global pointer_bug_pattern
pointer_bug_pattern:
    /* X0 holds a 64-bit pointer */
    add     w0, w0, #4      /* BUG: zero-extends; destroys upper
     ↪  address bits */
    ret
```

Correct pattern: pointer math must use X registers

```
.text
.global pointer_correct_pattern
```

```
pointer_correct_pattern:
    add     x0, x0, #4      /* correct 64-bit pointer increment */
    ret
```

Example: intentional truncation using W as a feature

Sometimes truncation and zero-extension is exactly what you want.

```
.text
.global truncate_to_u32
truncate_to_u32:
    /* X0 contains a value; return its low 32 bits, zero-extended */
    mov     w0, w0          /* canonical: keep low 32 bits, clear
     ↪  upper 32 */
    ret
```

Chapter 3

Special Registers and Their Roles

3.1 Stack Pointer (SP) — Rules and Restrictions

In AArch64, the stack pointer `SP` is an architectural register with dedicated semantics. It is not a normal general-purpose `Xn` register and cannot be treated as freely interchangeable with `X0--X30` in all instruction encodings.

Core rules (architectural discipline)

- **SP is used for stack addressing and stack adjustment.** It is the architectural anchor for stack frames and stack-based storage.
- **SP has encoding restrictions.** Many data-processing instructions only accept `Xn` registers; `SP` may be disallowed or may share encoding space with `XZR` depending on the instruction class.
- **SP is expected to remain suitably aligned.** Correct stack discipline assumes a consistent alignment policy; breaking alignment leads to faults or unpredictable

behavior in environments that enforce alignment.

- **SP must not be used as a general scratch register.** Use `Xn` registers for temporaries; reserve `SP` for stack operations.

Example 1: minimal stack allocation and deallocation

```
.text
.global stack_alloc_free
stack_alloc_free:
    sub     sp, sp, #32     /* allocate 32 bytes */
    add     sp, sp, #32     /* deallocate 32 bytes */
    ret
```

Example 2: saving and restoring registers on the stack

Paired store/load instructions are commonly used for efficient stack save/restore.

```
.text
.global save_restore_pair
save_restore_pair:
    sub     sp, sp, #16
    stp     x19, x20, [sp]  /* save two registers */
    ldp     x19, x20, [sp]  /* restore */
    add     sp, sp, #16
    ret
```

Example 3: stack object access via SP

```
.text
.global stack_local_u64
```

```
stack_local_u64:
    sub     sp, sp, #16
    mov     x1, #0x42
    str     x1, [sp, #8]    /* store local at SP+8 */
    ldr     x0, [sp, #8]    /* load local */
    add     sp, sp, #16
    ret
```

3.2 Link Register (LR / X30) — Call Semantics

AArch64 provides a dedicated architectural link register `LR` (alias `X30`). Call-like control-flow instructions set `LR` to the return address. The fundamental mechanism is:

> **Branch-with-link updates PC to the target and writes the return address into LR.**

Key implications

- **LR is live across nested calls.** If a function performs another call, the new call overwrites `LR`. Therefore, a function that calls other functions must preserve `LR` somewhere (typically on the stack or in a non-volatile register), depending on the chosen convention.
- **return typically branches to LR.** The architectural return pattern is a branch to the value in `LR`, commonly via `RET`.

Example 1: leaf function (no nested calls)

A leaf function can often rely on `LR` remaining intact.

```
.text
.global leaf_add
leaf_add:
    add     x0, x0, x1
    ret                     /* returns to LR */
```

Example 2: non-leaf function must preserve LR (non-ABI demonstration)

The following code shows the architectural necessity, independent of any ABI rule.

```
.text
.global nonleaf_demo
nonleaf_demo:
    sub     sp, sp, #16
    str     x30, [sp, #8]   /* preserve LR before nested call */

    bl      callee          /* overwrites LR */

    ldr     x30, [sp, #8]   /* restore LR */
    add     sp, sp, #16
    ret

.global callee
callee:
    add     x0, x0, #1
    ret
```

Example 3: explicit return via branch to LR

```
.text
.global explicit_return
explicit_return:
    br      x30             /* explicit branch to LR (RET is
     ↪  preferred) */
```

3.3 Program Counter (PC) — Architectural Behavior

The program counter `PC` is the architectural instruction address that drives instruction fetch. In AArch64:

- **PC is not a general-purpose register.** You do not treat it as `Xn`.
- **PC changes via control-flow instructions.** Sequential execution advances PC; branches/calls/returns modify PC.
- **PC-relative forms exist.** Certain instructions form addresses relative to the current instruction stream, enabling position-independent addressing.

Example 1: PC-relative data address formation pattern

```
.text
.global pc_relative_load
pc_relative_load:
    adrp    x0, my_data
    add     x0, x0, :lo12:my_data
    ldr     w0, [x0]
    ret

.data
.align 2
my_data:
    .word   7
```

3.4 Interaction Between PC and Branch Instructions

Branches are architectural PC-update operations. Understanding the branch family is essential:

Unconditional branch

```
.text
.global b_uncond
b_uncond:
    b       target
    mov     w0, #0          /* not executed */
target:
    mov     w0, #1
    ret
```

Conditional branch (PSTATE-driven)

Conditional branches test flags in PSTATE (set by compare or flag-setting arithmetic).

```
.text
.global b_cond
b_cond:
    cmp     w0, #0
    b.eq    is_zero
    mov     w0, #1
    ret
is_zero:
    mov     w0, #0
    ret
```

Branch with link (call semantics)

```
.text
.global call_demo
call_demo:
    bl      callee2         /* sets LR to return address and updates
     ↪  PC */
    add     w0, w0, #10
    ret

.global callee2
callee2:
    mov     w0, #5
    ret
```

Indirect branch (computed target)

```
.text
.global indirect_branch
indirect_branch:
    adr     x1, dest        /* compute address of dest into X1 */
    br      x1              /* jump to address in X1 */
    mov     w0, #0          /* not executed */
dest:
    mov     w0, #1
    ret
```

3.5 Register Usage Discipline (Non-ABI View)

This series separates architectural rules from ABI policy. At the architecture level, the discipline is:

Principles

- **Use `Xn` for addresses and pointer arithmetic.** Never use `Wn` for pointers unless you intentionally want truncation.
- **Treat `SP` as dedicated.** Use it only for stack allocation and stack-relative addressing.
- **Assume `LR` is volatile across any nested call.** Preserve it if your function calls another function.
- **Treat PC as control-flow state only.** Modify it only through branches, calls, and returns.
- **Explicitly track width.** Choose `W` vs `X` forms intentionally; do not mix by accident.

Example: pointer arithmetic must use X registers

```
.text
.global ptr_plus_8
ptr_plus_8:
    add     x0, x0, #8      /* correct pointer update */
    ret
```

Example: demonstrate the common pointer-width bug

```
.text
```

```
.global ptr_bug
ptr_bug:
    add     w0, w0, #8      /* BUG: truncates and zero-extends
     ↪  pointer */
    ret
```

Example: minimal non-leaf skeleton with LR preservation

```
.text
.global nonleaf_skeleton
nonleaf_skeleton:
    sub     sp, sp, #16
    str     x30, [sp, #8]   /* preserve LR */

    bl      worker          /* nested call overwrites LR */

    ldr     x30, [sp, #8]   /* restore LR */
    add     sp, sp, #16
    ret

.global worker
worker:
    add     w0, w0, #1
    ret
```

Chapter 4

PSTATE: Processor State Register

4.1 Conceptual Role of PSTATE

PSTATE is the architectural processor state in AArch64. It is not a general-purpose register and is not read/written as a single flat 64-bit value in normal code. Instead, its fields are updated by:

- arithmetic/logic instructions that set condition flags
- explicit system instructions (privileged or constrained) that modify control fields
- exception entry/return mechanisms that save/restore state as part of context changes

From an ISA programming perspective, the essential role of PSTATE is:

- **to carry condition flags** used for conditional branches and conditional selects
- **to carry execution control bits** that define how execution is masked or constrained
- **to provide architecturally defined state** that is consumed by control-flow and system behavior

Discipline rule

Treat PSTATE as **implicit architectural state**. You must explicitly track which instructions define the flags and which instructions merely consume them.

4.2 Condition Flags (N, Z, C, V)

AArch64 defines four primary condition flags in PSTATE:

- **N (Negative):** reflects the sign bit of the result (in the selected width).
- **Z (Zero):** set when the result is zero.
- **C (Carry):** indicates carry out for addition, and *no borrow* for subtraction.
- **V (Overflow):** indicates signed overflow in two's complement arithmetic.

Width rule (critical)

Flags are defined by the **instruction width**:

- `ADDS/SUBS/CMP` using `W` registers set flags as if operating on 32-bit values.
- `ADDS/SUBS/CMP` using `X` registers set flags as if operating on 64-bit values.

Example 1: Z flag for equality via CMP

```
.text
.global is_zero_u32
is_zero_u32:
    cmp     w0, #0          /* sets Z=1 iff W0 == 0 */
    cset    w0, eq          /* W0 = 1 if equal, else 0 */
    ret
```

Example 2: N flag from signed interpretation (width-dependent)

```
.text
```

```
.global is_negative_u32
is_negative_u32:
    cmp     w0, #0          /* signed compare uses flags from 32-bit
     ↪  subtraction */
    cset    w0, mi          /* W0 = 1 if negative (N==1), else 0 */
    ret
```

Example 3: C flag for unsigned comparison

For unsigned comparisons, the condition codes use C/Z in defined ways.

```
.text
.global u32_lt
u32_lt:
    /* returns 1 if W0 < W1 (unsigned), else 0 */
    cmp     w0, w1
    cset    w0, lo          /* LO means unsigned lower */
    ret
```

Example 4: V flag for signed overflow detection

```
.text
.global add_overflow_i32
add_overflow_i32:
    /* returns 1 if W0 + W1 overflows signed 32-bit, else 0 */
    adds    w2, w0, w1      /* sets V on signed overflow */
    cset    w0, vs          /* VS means overflow set */
    ret
```

4.3 Interrupt Mask Bits

PSTATE contains interrupt mask fields that control whether certain classes of interrupts are masked (blocked) while executing. These bits are part of the architectural state but are normally manipulated only in privileged contexts (OS, hypervisor, firmware). At user level, you usually **observe** their effects indirectly rather than modifying them.

Conceptual model

- Mask bits gate whether the CPU will take specific interrupt classes immediately.
- Masking does not "remove" interrupts; it defers their handling under architectural rules.
- Changes to masking are part of controlled system software design; application code must not rely on being able to change them.

Discipline rule

In this series, treat interrupt masks as **system-level control state**:

- understand that they exist and are part of PSTATE
- do not design user-level algorithms that depend on toggling them

4.4 Execution Control Bits

Beyond flags and interrupt masks, PSTATE includes execution control fields that affect how instructions execute or how exceptions are handled. These fields define constraints such as:

- whether single-step behavior is enabled (debug-related control)
- whether certain exception routing or execution constraints apply
- whether specific execution contexts are masked or controlled

Many of these bits are:

- privileged to modify, or
- only meaningful at specific exception levels, or
- architecturally updated as part of exception entry/return.

Discipline rule

For core AArch64 programming in this booklet:

- focus on **NZCV** as the primary consumable state for control flow
- treat the rest as **environment-defined** (OS/hypervisor controlled)

4.5 PSTATE vs CPSR (Conceptual Comparison)

In AArch32, CPSR is the classic named container for condition flags and control bits. In AArch64:

- PSTATE represents the architectural state conceptually filling that role.
- The programmer often interacts with specific subsets conceptually (e.g., NZCV flags) rather than treating the whole state as a single general register.

Key conceptual differences

- **Access style:** AArch64 emphasizes field-based semantics (flags/control) rather than direct "read/write CPSR" thinking.
- **Instruction set integration:** AArch64 provides rich conditional select and conditional branch usage that consumes NZCV.
- **Privilege separation:** many control fields are intentionally restricted to privileged contexts.

4.6 Instructions That Affect PSTATE

Flag-producing instructions

These instructions update NZCV (examples, not exhaustive):

- `CMP` (compare): subtracts operands and sets flags (no destination register)
- `CMN` (compare negative): adds operands and sets flags
- `TST` (test): ANDs operands and sets flags (no destination)
- `ADDS` / `SUBS`: arithmetic that writes a result and sets flags
- `ANDS`: logical AND that writes a result and sets flags

Example 1: CMP sets flags; CSET consumes them

```
.text
.global eq_u64
eq_u64:
    /* returns 1 if X0 == X1 else 0 */
    cmp     x0, x1
    cset    w0, eq
    ret
```

Example 2: SUBS sets flags for both signed and unsigned decisions

```
.text
.global classify_u32
classify_u32:
    /* returns 1 if W0 < 100 (unsigned), else 0 */
```

```
    subs    w1, w0, #100
    cset    w0, lo          /* LO uses C/Z derived from SUBS */
    ret
```

Example 3: TST/ANDS for bit tests

```
.text
.global has_bit3
has_bit3:
    /* returns 1 if bit 3 of W0 is set, else 0 */
    tst     w0, #(1 << 3)   /* sets Z if (W0 & mask)==0 */
    cset    w0, ne
    ret
```

Example 4: ANDS both computes and sets flags

```
.text
.global mask_and_check
mask_and_check:
    ands    w1, w0, #0xFF   /* W1 = W0 & 0xFF, sets NZCV based on W1
     ↪  */
    cset    w0, ne          /* W0 = 1 if W1 != 0 */
    ret
```

4.7 Common Flag-Related Pitfalls

Pitfall 1: Assuming an instruction sets flags when it does not

Only the `...S` forms (e.g., `ADDS`, `SUBS`, `ANDS`) update flags. Their non-`S` counterparts do not.

```
.text
.global pitfall_no_flags
pitfall_no_flags:
    add     w1, w0, #1      /* does NOT set flags */
    b.eq    was_zero        /* BUG: EQ tests old Z flag */
    mov     w0, #0
    ret
was_zero:
    mov     w0, #1
    ret
```

Correct pattern

```
.text
.global correct_flags
correct_flags:
    adds    w1, w0, #1      /* sets flags */
    b.eq    was_zero        /* now EQ is based on this result */
    mov     w0, #0
    ret
was_zero:
    mov     w0, #1
    ret
```

Pitfall 2: Mixing widths and misreading signedness

Flags are produced in the operation width. Using `W` when you intended `X` changes the computed flags.

```
.text
.global width_pitfall
width_pitfall:
    /* X0 contains a 64-bit value */
    cmp     w0, #0          /* flags computed from low 32 bits only
     ↪  */
    cset    w0, mi          /* result may be wrong for 64-bit intent
     ↪  */
    ret
```

Correct pattern for 64-bit sign test

```
.text
.global width_correct
width_correct:
    cmp     x0, #0
    cset    w0, mi
    ret
```

Pitfall 3: Confusing carry with overflow

C answers an **unsigned** carry/borrow question; **V** answers a **signed** overflow question. They are unrelated and must not be substituted.

Example: distinguish unsigned carry vs signed overflow

```
.text
.global carry_vs_overflow_demo
carry_vs_overflow_demo:
    adds    w2, w0, w1      /* sets both C and V as appropriate */
    cset    w3, cs          /* W3 = 1 if unsigned carry occurred */
    cset    w4, vs          /* W4 = 1 if signed overflow occurred */
    ret
```

Pitfall 4: Letting flags "leak" across unrelated code

Flags are global implicit state. If you branch based on flags, ensure the immediately preceding relevant instruction defines them.

```
.text
.global flag_leak_bug
flag_leak_bug:
    cmp     w0, #0
    add     w1, w1, #1      /* does NOT set flags */
    /* more code here could overwrite flags unexpectedly if it sets
     ↪ them */
    b.eq    is_zero
    mov     w0, #0
    ret
is_zero:
    mov     w0, #1
    ret
```

Correct discipline: define flags directly before consuming them

```
.text
.global flag_local_discipline
flag_local_discipline:
    /* ... unrelated code ... */
    cmp     w0, #0
    b.eq    is_zero
    mov     w0, #0
    ret
is_zero:
    mov     w0, #1
    ret
```

Chapter 5

Data Movement and Register Transfer

5.1 Register-to-Register Movement

AArch64 provides register-to-register movement for both 64-bit (`X`) and 32-bit (`W`) views. Conceptually, “move” copies bits from a source register to a destination register at the selected width.

Canonical forms

- `mov xD, xS` copies 64 bits.
- `mov wD, wS` copies 32 bits and, because it writes `wD`, it zero-extends into `xD`.
- Many `mov` forms are assembler aliases of logical operations (e.g., ORR with `XZR/WZR`) but the architectural intent is a pure register transfer.

Example 1: 64-bit move preserves full value

```
.text
```

```
.global mov_x_preserve
mov_x_preserve:
    mov     x0, x1          /* copy 64 bits from X1 to X0 */
    ret
```

Example 2: 32-bit move zero-extends into the X register

```
.text
.global mov_w_zero_ext
mov_w_zero_ext:
    mov     x2, -1          /* X2 = 0xFFFF_FFFF_FFFF_FFFF */
    mov     w2, w3          /* copy low 32 bits; X2 upper 32 cleared
     ↪  */
    ret
```

Example 3: explicit “copy” using the zero register

The assembler may accept a logical alias; the architectural meaning is a move.

```
.text
.global mov_via_zr
mov_via_zr:
    orr     x0, xzr, x1     /* X0 = X1 */
    ret
```

5.2 Immediate Encoding Constraints

AArch64 instructions are fixed 32-bit wide, so immediate fields are limited. As a result:

- not all integers fit as immediate operands in a single instruction,
- different instruction families support different immediate encodings,
- building a 64-bit constant is typically a multi-instruction sequence.

The architecture provides `MOVZ/MOVK/MOVN` for constructing immediates in 16-bit chunks at selected shift positions.

5.3 Zeroing vs Preserving Upper Bits

AArch64 makes a strict distinction between:

- **operations that fully define a register value** (overwrite/zero-fill),
- **operations that update only part of a register** (insert/keep other bits).

Two rules dominate:

1. Writing to `Wn` **always clears** the upper 32 bits of `Xn`.
2. `MOVZ/MOVN` produce a value defined by 16-bit fields and shift, with the rest **filled with zeros** (`MOVZ`) or ones (`MOVN`) in the selected width.

Example 1: fastest architectural zeroing patterns

```
.text
.global zero_patterns
zero_patterns:
    mov     x0, xzr          /* X0 = 0 */
    mov     w1, wzr          /* W1 = 0, and X1 upper cleared */
    ret
```

Example 2: W-write clears upper bits (pitfall or feature)

```
.text
.global clear_upper_by_w_write
clear_upper_by_w_write:
    mov     x0, #0x1122334455667788
    mov     w0, w0           /* keep low 32 bits, clear upper 32 */
    ret
```

5.4 MOV, MOVZ, MOVK, MOVN (Conceptual Use)

MOV (conceptual move / alias family)

`MOV` is used for:

- register-to-register copies,
- certain immediate moves that fit encodings,
- common zeroing idioms (`mov xD, xzr`).

MOVZ (Move Wide with Zero)

`MOVZ` writes a 16-bit immediate into a chosen 16-bit halfword lane and **zeros** the remaining bits (in the selected width). It is used to start constructing a constant.

```
.text
.global movz_example
movz_example:
    movz    x0, #0x1234, lsl #16   /* X0 = 0x0000_0000_1234_0000 */
    ret
```

MOVK (Move Wide with Keep)

`MOVK` writes a 16-bit immediate into a chosen halfword lane and **keeps** all other bits unchanged. It is used to *patch in* additional pieces of a constant.

```
.text
.global movk_patch_example
movk_patch_example:
```

```
    movz    x0, #0x1122, lsl #48
    movk    x0, #0x3344, lsl #32
    movk    x0, #0x5566, lsl #16
    movk    x0, #0x7788, lsl #0
    ret
```

MOVN (Move Wide with NOT)

`MOVN` writes the bitwise NOT of a 16-bit immediate into a chosen lane and fills the rest with ones (in the selected width). It is often useful for generating values with many one bits efficiently.

```
.text
.global movn_example
movn_example:
    movn    w0, #0x0        /* W0 = 0xFFFF_FFFF (all ones) */
    ret
```

Example: building -1 in 64-bit width

```
.text
.global minus_one_x
minus_one_x:
    movn    x0, #0x0        /* X0 = 0xFFFF_FFFF_FFFF_FFFF */
    ret
```

5.5 Register Width Interaction Rules

Width selection (`W` vs `X`) is part of the instruction semantics:

- `mov wD, wS` writes 32 bits and clears upper 32 bits of `xD`.
- `mov xD, xS` writes 64 bits and preserves full width.
- `movz/movk/movn` in `w` form define/patch only within 32-bit width and still clear upper bits via the write to `wD`.

Example: width changes the produced constant

```
.text
.global movn_width_demo
movn_width_demo:
    movn    w0, #0x0        /* W0 = 0xFFFF_FFFF, X0 becomes
     ↪  0x0000_0000_FFFF_FFFF */
    movn    x1, #0x0        /* X1 = 0xFFFF_FFFF_FFFF_FFFF */
    ret
```

5.6 Practical Register Transfer Patterns

Pattern 1: set register to zero (architectural)

```
.text
.global set_zero
set_zero:
    mov     x0, xzr          /* clear full 64-bit */
    ret
```

Pattern 2: copy pointer/address (always use X)

```
.text
.global copy_ptr
copy_ptr:
    mov     x1, x0           /* pointer copy */
    ret
```

Pattern 3: intentional truncation to 32 bits

```
.text
.global truncate_u32
truncate_u32:
    mov     w0, w0           /* keep low 32 bits, clear upper 32 */
    ret
```

Pattern 4: sign extension and zero extension (explicit)

Use explicit extension instructions when moving between widths and signedness.

```
.text
.global extend_examples
extend_examples:
    /* assume W0 holds a 32-bit value */
    uxtw    x1, w0          /* zero-extend W0 into X1 */
    sxtw    x2, w0          /* sign-extend W0 into X2 */
    ret
```

Pattern 5: load an address of a symbol (position-independent style)

```
.text
.global addr_of_symbol
addr_of_symbol:
    adrp    x0, symbol
    add     x0, x0, :lo12:symbol
    ret

.data
.align 3
symbol:
    .quad   0
```

Pattern 6: build a 64-bit constant (canonical wide-move sequence)

```
.text
.global build_constant_u64
build_constant_u64:
    movz    x0, #0xCAFE, lsl #48
    movk    x0, #0xBABE, lsl #32
    movk    x0, #0x1234, lsl #16
```

```
    movk    x0, #0x5678, lsl #0
    ret
```

Pattern 7: build a mask with many ones efficiently

```
.text
.global build_mask_many_ones
build_mask_many_ones:
    movn    x0, #0xFFFF, lsl #0   /* lower 16 bits become 0x0000,
     ↪  others ones */
    ret
```

Chapter 6

AArch64 Addressing Model

6.1 Load/Store Architecture Philosophy

AArch64 is a **load/store** architecture:

- **Loads** move data from memory into registers.
- **Stores** move data from registers into memory.
- **Arithmetic and logic** operate on registers, not directly on memory operands.

This separation is a core architectural discipline. It makes instruction behavior regular and keeps memory effects explicit and auditable.

Example: explicit memory access followed by register arithmetic

```
.text
.global load_compute_store
load_compute_store:
```

```
    ldr     x1, [x0]        /* load 64-bit from *X0 */
    add     x1, x1, #8      /* compute in registers */
    str     x1, [x0]        /* store 64-bit back to *X0 */
    ret
```

Example: what you do *not* do in AArch64

There is no "add memory, register" form like in x86. You must load first.

```
.text
.global not_x86_style
not_x86_style:
    /* AArch64 has no instruction that adds directly to a memory
     ↪  operand. */
    ret
```

6.2 Memory Access vs Register Operations

At the ISA level, each instruction belongs to one of two worlds:

- **Register world:** ALU, shifts, moves, compares, conditional selects.
- **Memory world:** `LDR/STR` and their size/extension variants, including paired forms.

A critical consequence is that memory ordering, faults, and alignment are tied to the **memory world** instructions only. Pure register instructions do not access memory and cannot fault due to memory address issues.

Example: one memory access, multiple register operations

```
.text
.global one_load_many_ops
one_load_many_ops:
    ldr     w1, [x0]        /* one memory read */
    add     w1, w1, #1
    eor     w1, w1, #0x5A
    lsl     w1, w1, #2
    str     w1, [x0]        /* one memory write */
    ret
```

Example: paired load/store emphasize explicit memory effects

```
.text
.global pair_mem_ops
pair_mem_ops:
    ldp     x1, x2, [x0]    /* load two 64-bit values */
    add     x1, x1, x2      /* compute */
```

```
    stp     x1, x2, [x0]    /* store back */
    ret
```

6.3 Address Calculation Model

AArch64 forms an **effective address (EA)** for each memory access. Conceptually:

EA = Base + Offset

Where:

- the **base** is typically an `Xn` register (or `SP` for stack addressing),
- the **offset** may be an immediate, a register (optionally shifted/extended), or implied by an addressing mode.

This booklet treats EA computation as a pure architectural step: compute the address first, then perform the memory access at that address.

Example 1: immediate offset addressing (EA = X0 + imm)

```
.text
.global ea_imm_offset
ea_imm_offset:
    ldr     w1, [x0, #12]    /* EA = X0 + 12 */
    ret
```

Example 2: register offset addressing (EA = X0 + X1)

```
.text
.global ea_reg_offset
ea_reg_offset:
    ldr     w2, [x0, x1]     /* EA = X0 + X1 */
    ret
```

Example 3: scaled indexing for element access

Scaled addressing is used to index elements by size (e.g., 4 bytes for 32-bit, 8 bytes for 64-bit).

```
.text
.global ea_scaled_u64
ea_scaled_u64:
    /* load X0 = *(base + (index<<3)) for 8-byte elements */
    ldr     x0, [x1, x2, lsl #3]   /* EA = X1 + (X2 * 8) */
    ret
```

Example 4: SP as base for stack-resident data

```
.text
.global ea_sp_local
ea_sp_local:
    sub     sp, sp, #16
    mov     x1, #0x42
    str     x1, [sp, #8]    /* EA = SP + 8 */
    ldr     x0, [sp, #8]
    add     sp, sp, #16
    ret
```

6.4 Alignment Rules and Enforcement

Alignment is an architectural property of memory access:

- Many loads/stores have a **natural alignment** expectation: e.g., 8-byte loads prefer 8-byte aligned addresses.
- The architecture can **enforce** alignment (raising an exception) depending on system configuration and the type of access.
- Even when misaligned access is permitted by the environment, it may be slower and may interact poorly with atomicity requirements.

Discipline rule

For correct and portable low-level code:

- assume **natural alignment is required** for the data size,
- treat misalignment as a bug unless you explicitly designed for it and verified system behavior.

Example: natural alignment for 64-bit data

```
.text
.global aligned_u64_access
aligned_u64_access:
    /* assume X0 is 8-byte aligned */
    ldr     x1, [x0]        /* aligned 8-byte load */
    str     x1, [x0, #8]    /* aligned 8-byte store */
    ret
```

Example: alignment bug pattern (conceptual)

The following illustrates a dangerous pattern: forcing an odd address then performing a wide load/store.

```
.text
.global misaligned_bug_pattern
misaligned_bug_pattern:
    add     x0, x0, #1      /* likely breaks natural alignment */
    ldr     x1, [x0]        /* may fault or be slow depending on
     ↪  enforcement */
    ret
```

Example: structure layout discipline

Use correct alignment directives for data objects so that code relying on natural alignment remains valid.

```
.data
.align 3                    /* 2^3 = 8-byte alignment */
global_u64:
    .quad   0x1122334455667788
```

6.5 Address Size and Virtual Address Space (Conceptual)

AArch64 uses 64-bit registers for addresses, but the architecture defines that not all 64 bits must be used as meaningful address bits in a given configuration. The effective virtual address size is an architectural/system choice.

Conceptual model (what matters for core programming)

- **Pointers live in X registers.** Address computations must use `Xn`, never `Wn`, unless you intentionally truncate.
- **Virtual addresses are interpreted by the execution environment.** Translation, permissions, and faults are system-controlled (OS/hypervisor/MMU).
- **Not all 64-bit patterns are valid addresses.** Some bit patterns may be invalid or fault when used as addresses, depending on configured address size and translation rules.

Example: pointer arithmetic must remain 64-bit

```
.text
.global ptr_arith_correct
ptr_arith_correct:
    /* X0 is a pointer */
    add     x0, x0, #32     /* correct: preserves full address */
    ret
```

Example: pointer truncation bug (do not do this)

```
.text
.global ptr_truncation_bug
```

```
ptr_truncation_bug:
    add     w0, w0, #32     /* BUG: zero-extends; destroys upper
     ↪  address bits */
    ret
```

Example: PC-relative addressing stays within architectural rules

PC-relative forms generate addresses in a way that is compatible with position-independent layout.

```
.text
.global pc_relative_addressing
pc_relative_addressing:
    adrp    x0, symbol
    add     x0, x0, :lo12:symbol
    ret

.data
.align 3
symbol:
    .quad   0
```

Chapter 7

Addressing Modes in AArch64

7.1 Immediate Offset Addressing

Immediate offset addressing forms an effective address by adding a constant offset to a base register:

EA = Base + Imm

The base is typically an `Xn` register (or `SP` for stack accesses). Immediate offsets are constrained by the instruction encoding; many load/store forms support a range of offsets, often with alignment-related scaling rules.

Example 1: element access with immediate offset

```
.text
.global imm_offset_u32
imm_offset_u32:
    /* load element 3 (4 bytes each): EA = base + 12 */
```

```
    ldr     w0, [x0, #12]
    ret
```

Example 2: stack local access (SP as base)

```
.text
.global sp_local_access
sp_local_access:
    sub     sp, sp, #32
    str     x0, [sp, #16]   /* EA = SP + 16 */
    ldr     x0, [sp, #16]
    add     sp, sp, #32
    ret
```

7.2 Register Offset Addressing

Register offset addressing forms the effective address by adding a register value (optionally shifted/extended) to a base:

EA = Base + (Index [shift/extend])

This mode is used for dynamic indexing, pointer chasing, and table lookups.

Example 1: basic register offset

```
.text
.global reg_offset_basic
reg_offset_basic:
    /* EA = X0 + X1 */
    ldr     w0, [x0, x1]
    ret
```

Example 2: array indexing with scaled register (8-byte elements)

```
.text
.global reg_offset_scaled_u64
reg_offset_scaled_u64:
    /* EA = base + (index * 8) */
    ldr     x0, [x1, x2, lsl #3]
    ret
```

7.3 Scaled and Unscaled Offsets

AArch64 distinguishes between **scaled** and **unscaled** immediate offsets.

Scaled immediate offsets

Many `LDR/STR` forms use a scaled immediate:

- the encoded immediate represents *units of the access size*
- the effective byte offset is $Imm \times access\_size$

This makes common aligned accesses compact in encoding.

Unscaled immediate offsets

Unscaled forms use a byte offset directly (often signed and smaller range). These are used when you need small negative offsets or byte-precise control.

Example 1: scaled offset concept for 64-bit load

```
.text
.global scaled_imm_concept
scaled_imm_concept:
    /* typical aligned access: base + 16 bytes */
    ldr     x0, [x1, #16]
    ret
```

Example 2: unscaled offset using LDUR/STUR family

Unscaled forms allow byte-granular signed offsets.

```
.text
.global unscaled_signed_offset
unscaled_signed_offset:
    /* access at base - 8 bytes */
    ldur    x0, [x1, #-8]
    ret
```

Example 3: scaled vs unscaled when indexing structure fields

```
.text
.global struct_field_access
struct_field_access:
    /* assume a structure at X0; field at +24 */
    ldr     x1, [x0, #24]   /* aligned field access */
    ret
```

7.4 Pre-Indexed and Post-Indexed Addressing

These addressing modes combine memory access with base register update. They are essential for stack operations and pointer-walking loops.

Pre-indexed

Base := Base + Imm; EA := Base; then memory access

Post-indexed

EA := Base; memory access; then Base := Base + Imm

Example 1: stack push/pop style with pre-indexed

```
.text
.global push_pop_lr
push_pop_lr:
    str     x30, [sp, #-16]!    /* pre-index: SP = SP - 16; store at
    ↪  new SP */
    ldr     x30, [sp], #16      /* post-index: load at SP; then SP =
    ↪  SP + 16 */
    ret
```

Example 2: walking through an array with post-indexed loads

```
.text
.global sum_two_u64_post
sum_two_u64_post:
    /* X0 = base pointer, returns X0 = a[0] + a[1] */
```

```
    ldr     x1, [x0], #8        /* load a[0], then advance pointer */
    ldr     x2, [x0], #8        /* load a[1], then advance pointer */
    add     x0, x1, x2
    ret
```

Example 3: repeated stores with post-indexed addressing

```
.text
.global fill_two_u32
fill_two_u32:
    /* X0 = pointer, W1 = value; store twice and advance */
    str     w1, [x0], #4
    str     w1, [x0], #4
    ret
```

7.5 Common Addressing Mode Mistakes

Mistake 1: Using W registers for addresses

Addresses must live in `X` registers. Using `Wn` for address arithmetic truncates and zero-extends, destroying upper address bits.

```
.text
.global addr_bug_w_reg
addr_bug_w_reg:
    add     w0, w0, #8      /* BUG: corrupts pointer */
    ldr     x1, [x0]
    ret
```

Correct pattern

```
.text
.global addr_ok_x_reg
addr_ok_x_reg:
    add     x0, x0, #8
    ldr     x1, [x0]
    ret
```

Mistake 2: Confusing pre-index vs post-index

Pre-index changes the base *before* the access; post-index changes it *after*. Mixing them breaks stack and pointer-walk logic.

```
.text
.global pre_post_confusion
```

```
pre_post_confusion:
    /* intended: load then advance; but uses pre-index (advances
     ↪  first) */
    ldr     x1, [x0, #8]!   /* pointer advanced before load */
    ret
```

Mistake 3: assuming offsets are always byte offsets

Some forms scale immediates by access size; others are unscaled. When exact byte control is required (especially negative offsets), use unscaled forms.

```
.text
.global need_byte_precise_negative
need_byte_precise_negative:
    /* conceptual: access base - 1 (byte) */
    ldurb   w0, [x1, #-1]   /* byte-granular negative offset */
    ret
```

Mistake 4: breaking alignment accidentally

Indexing mistakes (wrong scale, wrong offset) often produce misalignment. Treat misalignment as a correctness bug unless explicitly designed and validated.

```
.text
.global alignment_bug_pattern
alignment_bug_pattern:
    add     x0, x0, #2      /* likely breaks 4/8-byte alignment */
    ldr     x1, [x0]        /* may fault or degrade performance */
    ret
```

7.6 Addressing Mode Selection Discipline

Use this selection discipline to avoid subtle bugs and keep code audit-friendly:

Rule 1: choose addressing to match the access pattern

- **Fixed field access:** prefer immediate offsets (`[base, #imm]`).
- **Array indexing:** prefer scaled register offsets (`[base, index, lsl #k]`).
- **Pointer walking:** prefer post-indexed forms (`[base], #imm`) for clarity.
- **Stack push/pop:** use pre-indexed store and post-indexed load patterns carefully.

Rule 2: keep pointer arithmetic in X registers

Never compute addresses in `W` registers.

Rule 3: isolate flag-dependent code from addressing

Addressing mode updates (pre/post-index) modify base registers; keep them readable and local to avoid mixing with flag-sensitive control flow.

Example: disciplined array loop (post-indexed, aligned)

```
.text
.global sum_four_u32
sum_four_u32:
    /* X0 = base, returns W0 = sum of 4 elements */
    mov     w3, wzr
```

```
    ldr     w1, [x0], #4
    add     w3, w3, w1
    ldr     w1, [x0], #4
    add     w3, w3, w1
    ldr     w1, [x0], #4
    add     w3, w3, w1
    ldr     w1, [x0], #4
    add     w3, w3, w1

    mov     w0, w3
    ret
```

Example: disciplined stack save/restore (pre/post-index)

```
.text
.global save_restore_x19_x20
save_restore_x19_x20:
    stp     x19, x20, [sp, #-16]!   /* push */
    /* ... body ... */
    ldp     x19, x20, [sp], #16     /* pop */
    ret
```

Chapter 8

Load and Store Instruction Behavior

8.1 Basic Load/Store Semantics

AArch64 is a load/store ISA: memory is accessed only through explicit load and store instructions. Each load/store performs:

1. **effective address (EA) calculation** from base + offset (and optional update),
2. **a memory read or write** of a defined size,
3. **a register writeback** for loads (with defined extension behavior) or a memory write for stores.

Loads and stores are **size-specific**. The access size is part of the instruction semantics and controls:

- how many bytes are transferred,
- which bits are written in the destination register (loads),
- whether extension (sign/zero) is applied.

Example: classic read-modify-write

```
.text
.global rmw_u64
rmw_u64:
    ldr     x1, [x0]        /* load 8 bytes from [X0] into X1 */
    add     x1, x1, #1
    str     x1, [x0]        /* store 8 bytes back to [X0] */
    ret
```

Example: load with post-index (pointer walking)

```
.text
.global load_walk_u32
load_walk_u32:
    ldr     w1, [x0], #4    /* load 4 bytes; then X0 += 4 */
    ldr     w2, [x0], #4
    add     w0, w1, w2
    ret
```

8.2 Byte, Halfword, Word, and Doubleword Access

AArch64 provides load/store forms for different element sizes:

- **Byte (8-bit):** `LDRB` / `STRB`
- **Halfword (16-bit):** `LDRH` / `STRH`
- **Word (32-bit):** `LDR` (to `W`) / `STR` (from `W`)
- **Doubleword (64-bit):** `LDR` (to `X`) / `STR` (from `X`)

Example: store and load each size

```
.text
.global size_access_demo
size_access_demo:
    /* X0 = base address */

    mov     w1, #0xAA
    strb    w1, [x0, #0]    /* store 1 byte */

    mov     w1, #0x1234
    strh    w1, [x0, #2]    /* store 2 bytes */

    mov     w1, #0x89ABCDEF
    str     w1, [x0, #4]    /* store 4 bytes */

    mov     x1, #0x1122334455667788
    str     x1, [x0, #8]    /* store 8 bytes */
```

```
    ldrb    w2, [x0, #0]    /* load 1 byte into W2 (zero-extended) */
    ldrh    w3, [x0, #2]    /* load 2 bytes into W3 (zero-extended)
     ↪  */
    ldr     w4, [x0, #4]    /* load 4 bytes into W4 (zero-extended
     ↪  into X4) */
    ldr     x5, [x0, #8]    /* load 8 bytes into X5 */

    ret
```

8.3 Signed vs Unsigned Loads

For sub-32-bit loads (byte/halfword) and for many 32-bit-to-64-bit loads, AArch64 offers explicit signed and unsigned variants:

- **Unsigned loads** zero-extend the loaded value.
- **Signed loads** sign-extend the loaded value to the destination width.

Common signed-load mnemonics:

- `LDRSB` load signed byte (extend to 32 or 64 depending on destination)
- `LDRSH` load signed halfword
- `LDRSW` load signed word and sign-extend to 64-bit (`X` destination)

Example: unsigned byte vs signed byte

```
.text
.global signed_unsigned_byte
signed_unsigned_byte:
    /* memory byte at [X0] is treated either as unsigned or signed */
    ldrb    w1, [x0]        /* W1 = zero-extended 0..255 */
    ldrsb   x2, [x0]        /* X2 = sign-extended -128..127 */
    ret
```

Example: load signed 32-bit into 64-bit using LDRSW

```
.text
.global load_i32_to_i64
load_i32_to_i64:
```

```
    ldrsw   x0, [x0]          /* X0 = sign-extended 32-bit value from
    ↪  memory */
    ret
```

8.4 Zero-Extension vs Sign-Extension

Extension behavior is a core part of load semantics:

- **Zero-extension** fills upper bits with zeros.
- **Sign-extension** copies the sign bit into upper bits.

Two architecture-critical rules:

1. Loads into `Wn` write 32 bits and therefore **clear upper 32 bits of `Xn`** (by the W-write rule).
2. Signed-load variants that target `Xn` produce a full 64-bit sign-extended value.

Example: load 32-bit unsigned vs signed into 64-bit

```
.text
.global load_u32_vs_i32
load_u32_vs_i32:
    /* X0 = address */

    ldr     w1, [x0]        /* unsigned interpretation: X1 =
     ↪  0x00000000XXXXXXXX */
    ldrsw   x2, [x0]        /* signed interpretation:   X2 =
     ↪  sign-extended */

    ret
```

Example: explicit extension after an unsigned sub-word load

```
.text
```

```
.global extend_after_ldrb
extend_after_ldrb:
    ldrb    w0, [x0]        /* W0 = 0..255 */
    /* optional: treat as signed byte after load */
    sxtb    w0, w0          /* sign-extend low 8 bits within 32-bit
     ↪  */
    ret
```

8.5 Paired Load/Store Instructions (LDP / STP)

`LDP` and `STP` transfer two registers with one instruction and use a single addressing mode. They are commonly used for:

- saving/restoring registers on the stack,
- copying small blocks,
- efficient function prolog/epilog patterns.

Example: stack save/restore using paired access

```
.text
.global save_restore_pair
save_restore_pair:
    stp     x19, x20, [sp, #-16]!  /* push */
    /* ... body ... */
    ldp     x19, x20, [sp], #16    /* pop */
    ret
```

Example: load two adjacent 64-bit elements

```
.text
.global load_two_u64
load_two_u64:
    /* X0 = base pointer */
    ldp     x1, x2, [x0]    /* loads *(X0+0) into X1 and *(X0+8) into
    ↪  X2 */
    add     x0, x1, x2
    ret
```

Example: store two registers to memory with an immediate offset

```
.text
.global store_two_u64
store_two_u64:
    /* X0 = base pointer */
    stp     x1, x2, [x0, #16]  /* stores at base+16 and base+24 */
    ret
```

8.6 Architectural Guarantees and Limitations

What the ISA guarantees at the core level

- **Precise access size:** each load/store transfers exactly the size encoded by the instruction.
- **Defined extension behavior:** signed/unsigned variants and destination width define the result bits.
- **Explicit memory effects:** only load/store instructions access memory.
- **Addressing mode semantics:** pre/post-index updates are architecturally defined.

What the ISA does not guarantee (environment-defined)

- **Whether misaligned access faults or works:** alignment enforcement depends on system configuration and access type.
- **Atomicity of ordinary loads/stores beyond architectural rules:** multi-byte transfers may not be atomic with respect to concurrency unless the architecture and system guarantee it for the size and alignment.
- **Ordering between different memory operations:** memory ordering is governed by the architecture's memory model and specific barrier/atomic instructions (covered in later booklets).

Example: misalignment as a correctness bug pattern

```
.text
.global misalignment_risk
```

```
misalignment_risk:
    add     x0, x0, #1      /* likely breaks 8-byte alignment */
    ldr     x1, [x0]        /* may fault or be slow depending on
     ↪  enforcement */
    ret
```

Example: keep alignment and size consistent

```
.text
.global aligned_pair_load
aligned_pair_load:
    /* assume X0 is 16-byte aligned for paired 64-bit access */
    ldp     x1, x2, [x0]    /* aligned load of 16 bytes total */
    ret
```

Example: select the correct signed/unsigned load

```
.text
.global choose_load_variant
choose_load_variant:
    /* X0 points to an 8-bit element */
    ldrb    w1, [x0]        /* treat as unsigned */
    ldrsb   x2, [x0]        /* treat as signed */
    ret
```

Chapter 9

Addressing, Registers, and Performance (Conceptual)

9.1 Instruction Count vs Addressing Choice

In AArch64, addressing mode choice directly affects instruction count because loads/stores can incorporate:

- immediate offsets,
- scaled register offsets,
- pre/post-index base updates.

The architectural goal is not "micro-optimization" but **expressing the access pattern in fewer instructions without hiding meaning**. Fewer instructions can reduce pressure on fetch/decode and can simplify dependency chains, but correctness and clarity remain primary.

Example 1: separate pointer update vs post-index (same intent)

Two-instruction pattern (explicit update):

```
.text
.global load_then_add_ptr
load_then_add_ptr:
    ldr     w1, [x0]        /* load */
    add     x0, x0, #4      /* advance pointer */
    ret
```

One-instruction pattern (post-index update):

```
.text
.global load_post_index
load_post_index:
    ldr     w1, [x0], #4    /* load then advance pointer */
    ret
```

Example 2: scaled register index reduces extra multiply/add

```
.text
.global scaled_index_u64
scaled_index_u64:
    /* load a[index] where elements are 8 bytes */
    ldr     x0, [x1, x2, lsl #3]    /* EA = base + index*8 */
    ret
```

9.2 Register Pressure and Access Patterns

Register pressure is the demand for live registers at once. In AArch64, pressure is shaped by:

- the number of simultaneously live pointers, indices, accumulators, temporaries,
- how long values stay live between load and use,
- whether addressing mode choice reduces temporaries (e.g., avoiding a separate “address register”).

Example 1: avoid extra address temporaries

More pressure (extra address register):

```
.text
.global extra_addr_temp
extra_addr_temp:
    add     x3, x0, x1      /* X3 = base + offset */
    ldr     w2, [x3]        /* load via temporary address register */
    ret
```

Less pressure (use register offset addressing):

```
.text
.global no_addr_temp
no_addr_temp:
    ldr     w2, [x0, x1]    /* EA = X0 + X1, no extra register */
    ret
```

Example 2: shorten live ranges to reduce pressure

```
.text
.global short_live_range
short_live_range:
    ldr     x1, [x0]        /* load */
    add     x1, x1, #1      /* consume soon */
    str     x1, [x0]        /* store back */
    ret
```

Example 3: pointer-walking loop with minimal live registers

```
.text
.global sum_four_u32_min_regs
sum_four_u32_min_regs:
    /* X0 = base pointer; returns W0 = sum of 4 u32 */
    mov     w2, wzr         /* accumulator */

    ldr     w1, [x0], #4
    add     w2, w2, w1
    ldr     w1, [x0], #4
    add     w2, w2, w1
    ldr     w1, [x0], #4
    add     w2, w2, w1
    ldr     w1, [x0], #4
    add     w2, w2, w1

    mov     w0, w2
    ret
```

9.3 Alignment Impact on Execution

Alignment affects both correctness (possible faults) and performance (possible extra work). AArch64 defines alignment expectations for many access sizes, and the **execution environment** may:

- permit misaligned accesses with potential penalties,
- or enforce alignment by raising an exception for misaligned access,
- or impose stricter rules for certain operations (e.g., atomics) than for ordinary loads/stores.

Discipline rule

For predictable and portable code:

- maintain **natural alignment** for the access size,
- treat misalignment as a bug unless explicitly designed and validated for the target environment.

Example: aligned 64-bit sequential access

```
.text
.global aligned_sequential_u64
aligned_sequential_u64:
    /* assume X0 is 8-byte aligned */
    ldr     x1, [x0]
    ldr     x2, [x0, #8]
    add     x0, x1, x2
    ret
```

Example: misalignment bug pattern

```
.text
.global misalignment_bug
misalignment_bug:
    add     x0, x0, #1      /* likely breaks 8-byte alignment */
    ldr     x1, [x0]        /* may fault or incur penalty */
    ret
```

Example: enforce alignment in data layout

```
.data
.align 3
aligned_qword:
    .quad   0x1122334455667788
```

9.4 Addressing Discipline for Predictable Code

Performance-friendly code is usually the result of **predictable addressing** and **clear dependency chains**, not exotic tricks. The discipline below yields code that is easier for humans and tools to reason about:

Discipline rules

1. **Keep pointers in X registers only.** Never compute addresses in W registers.
2. **Prefer one canonical style per pattern:**
   - structure field access: `[base, #imm]`
   - array indexing: `[base, index, lsl #k]`
   - pointer walking: `[base], #imm` (post-index) or explicit `add`
   - stack access: `[sp, #imm]` with disciplined allocation
3. **Avoid mixing flag-dependent control flow with base-update addressing.** Keep conditional branches close to the flag-setting instruction and keep pointer updates readable.
4. **Keep effective address computation obvious.** If EA is not obvious in one glance, prefer a clearer sequence.

Example: correct pointer arithmetic uses X

```
.text
.global ptr_increment_ok
ptr_increment_ok:
    add     x0, x0, #16     /* correct pointer update */
```

```
    ret
```

Example: incorrect pointer arithmetic uses W (catastrophic on real systems)

```
.text
.global ptr_increment_bug
ptr_increment_bug:
    add     w0, w0, #16     /* BUG: truncates then zero-extends
     ↪  address */
    ret
```

9.5 What the ISA Guarantees — and What It Does Not

What the ISA guarantees

- **Exact semantics of addressing modes:** how EA is formed and when base writeback occurs.
- **Exact access size and extension behavior:** loads/stores transfer a defined number of bytes and define destination register bits accordingly.
- **Defined register-width interactions:** W writes clear upper 32 bits of the corresponding X register.

What the ISA does not guarantee (depends on microarchitecture and environment)

- **Cycle cost of any sequence:** decode width, pipeline depth, cache behavior, and memory subsystem vary across cores.
- **Whether misaligned access faults or is handled transparently:** can depend on system configuration and access type.
- **Atomicity and ordering of ordinary loads/stores under concurrency:** requires the memory model, barriers, and atomic instructions (handled in later booklets).

Example: writeback is guaranteed, performance is not

Post-index semantics are guaranteed; the speed depends on the core.

```
.text
.global writeback_semantics
```

```
writeback_semantics:
    ldr     w1, [x0], #4    /* guaranteed: X0 updates after the load
     ↪  */
    ldr     w2, [x0], #4
    add     w0, w1, w2
    ret
```

Example: keep performance reasoning separate from correctness

The following forms are architecturally correct; which is faster is not an ISA guarantee.

```
.text
.global same_semantics_two_forms
same_semantics_two_forms:
    /* form A: explicit update */
    ldr     w1, [x0]
    add     x0, x0, #4

    /* form B: post-index update */
    ldr     w2, [x0], #4

    add     w0, w1, w2
    ret
```

Chapter 10

Common Errors and Dangerous Assumptions

10.1 Misusing W Registers in 64-bit Contexts

In AArch64, `Wn` is the low 32-bit view of `Xn`. The architectural rule is:

> **Any write to a W register clears the upper 32 bits of the corresponding X register.**

This rule is a frequent source of catastrophic bugs when `X` registers hold pointers, 64-bit counters, or 64-bit masks.

Bug pattern 1: pointer arithmetic in W

```
.text
.global bug_ptr_w
bug_ptr_w:
```

```
    /* X0 holds a 64-bit pointer */
    add     w0, w0, #8      /* BUG: truncates and zero-extends
     ↪  pointer */
    ldr     x1, [x0]        /* may fault or read wrong memory */
    ret
```

Correct pattern

```
.text
.global ok_ptr_x
ok_ptr_x:
    add     x0, x0, #8      /* correct 64-bit pointer update */
    ldr     x1, [x0]
    ret
```

Bug pattern 2: destroying upper bits of a 64-bit mask

```
.text
.global bug_mask_w_write
bug_mask_w_write:
    mov     x0, #0xFFFF000000000000
    mov     w0, w0          /* BUG: clears upper 32 bits, destroys
     ↪  the mask */
    ret
```

Correct pattern: preserve full width

```
.text
.global ok_mask_x
ok_mask_x:
```

```
    /* keep mask in X registers and use X-form operations */
    and     x0, x0, x1
    ret
```

Intentional truncation (safe use of W)

Using `Wn` is correct when you explicitly intend 32-bit semantics and want zero-extension.

```
.text
.global intentional_trunc_u32
intentional_trunc_u32:
    mov     w0, w0          /* keep low 32 bits, clear upper 32 */
    ret
```

10.2 Assuming PC Is Directly Writable

In AArch64, `PC` is not a general-purpose register. You do not write it with normal data-processing instructions. Control flow changes `PC` through:

- direct branches (`b`, `b.cond`, `bl`)
- indirect branches (`br`, `blr`)
- returns (`ret`)

Bug assumption: "add to PC" style reasoning

AArch64 does not support treating PC as `Xn` in ordinary arithmetic.

```
.text
.global bug_pc_writable_assumption
bug_pc_writable_assumption:
    /* There is no correct "add pc, pc, #imm" data-processing model
     ↪  like a GPR. */
    ret
```

Correct: express control flow with branches

```
.text
.global ok_branch_flow
ok_branch_flow:
    b       target
    mov     w0, #0          /* not executed */
target:
    mov     w0, #1
    ret
```

Correct: computed control flow uses BR/BLR

```
.text
.global ok_indirect_branch
ok_indirect_branch:
    adr     x1, dest
    br      x1
    mov     w0, #0          /* not executed */
dest:
    mov     w0, #1
    ret
```

10.3 Confusing SP with General Registers

`SP` is a special register. It is used for stack allocation and stack-relative addressing and has encoding restrictions. It is not interchangeable with `Xn` in all instruction classes, and it must obey a disciplined alignment policy.

Bug pattern: using SP as a scratch register

```
.text
.global bug_sp_scratch
bug_sp_scratch:
    /* DO NOT treat SP as a free temporary register */
    add     sp, sp, #1      /* breaks alignment and stack discipline
     ↪  */
    ret
```

Correct: use SP only for stack adjustment

```
.text
.global ok_sp_adjust
ok_sp_adjust:
    sub     sp, sp, #16     /* allocate */
    add     sp, sp, #16     /* deallocate */
    ret
```

Correct: save/restore using disciplined SP updates

```
.text
.global ok_sp_save_restore
ok_sp_save_restore:
```

```
    stp     x19, x20, [sp, #-16]!   /* push */
    /* ... */
    ldp     x19, x20, [sp], #16     /* pop */
    ret
```

10.4 Flag Dependency Bugs

NZCV flags in PSTATE are implicit state. Bugs occur when code assumes:

- a non-flag-setting instruction set flags,
- flags remain meaningful across unrelated operations,
- width does not matter for flag computation.

Bug pattern 1: using ADD instead of ADDS

```
.text
.global bug_missing_flags
bug_missing_flags:
    add     w1, w0, #1      /* does NOT set flags */
    b.eq    was_zero        /* BUG: tests old Z flag */
    mov     w0, #0
    ret
was_zero:
    mov     w0, #1
    ret
```

Correct pattern

```
.text
.global ok_flags
ok_flags:
    adds    w1, w0, #1      /* sets flags */
    b.eq    was_zero
    mov     w0, #0
```

```
    ret
was_zero:
    mov     w0, #1
    ret
```

Bug pattern 2: width mismatch in compare

```
.text
.global bug_cmp_width
bug_cmp_width:
    /* X0 holds a 64-bit value */
    cmp     w0, #0          /* flags computed from low 32 bits only
     ↪  */
    b.mi    neg             /* may be wrong for 64-bit intent */
    mov     w0, #0
    ret
neg:
    mov     w0, #1
    ret
```

Correct: compare in the intended width

```
.text
.global ok_cmp_width
ok_cmp_width:
    cmp     x0, #0
    b.mi    neg
    mov     w0, #0
    ret
neg:
```

```
    mov     w0, #1
    ret
```

10.5 Addressing Mode Miscalculations

Most addressing bugs are EA bugs: computing the wrong effective address due to:

- wrong scale factor,
- confusion between byte offset and element offset,
- mixing pre-index and post-index semantics,
- base corruption (often from W writes).

Bug pattern 1: wrong scale for element size

```
.text
.global bug_wrong_scale_u64
bug_wrong_scale_u64:
    /* X0 = base, X1 = index, elements are 8 bytes */
    ldr     x2, [x0, x1, lsl #2]    /* BUG: uses *4 instead of *8 */
    ret
```

Correct pattern

```
.text
.global ok_scale_u64
ok_scale_u64:
    ldr     x2, [x0, x1, lsl #3]    /* correct: index * 8 */
    ret
```

Bug pattern 2: pre-index vs post-index confusion

```
.text
```

```
.global bug_pre_vs_post
bug_pre_vs_post:
    /* intended: load then advance pointer by 8 */
    ldr     x1, [x0, #8]!   /* BUG: advances before load */
    ret
```

Correct pattern: post-index

```
.text
.global ok_post_index
ok_post_index:
    ldr     x1, [x0], #8    /* load then advance */
    ret
```

Bug pattern 3: using byte offset when index is in elements (or vice versa)

```
.text
.global bug_element_vs_byte
bug_element_vs_byte:
    /* X1 is element index, but used as byte offset */
    ldr     w0, [x0, x1]    /* BUG unless each element is 1 byte */
    ret
```

Correct: explicitly scale by element size

```
.text
.global ok_element_index_u32
ok_element_index_u32:
    ldr     w0, [x0, x1, lsl #2]    /* element index * 4 */
    ret
```

10.6 Debugging Register and Addressing Errors

When debugging AArch64 register/addressing bugs, use a disciplined, architecture-first method:

Checklist: register-width and aliasing

1. Search for any `W`-writes to registers that later act as pointers (`X`).
2. Verify pointer arithmetic uses `X` forms (`add xN, ...`), not `W`.
3. Confirm extension intent: unsigned loads (`LDRB/LDRH`) vs signed loads (`LDRSB/LDRSH/LDRSW`).

Checklist: effective address validation

1. For each memory instruction, write EA explicitly as **Base + Offset**.
2. Confirm whether the offset is scaled or unscaled and whether it is byte-granular.
3. If pre/post-index is used, track when the base is updated relative to the access.

Checklist: flags and control flow

1. Identify the exact instruction that sets the flags used by each conditional branch.
2. Ensure no unintended flag-setting instruction sits between flag definition and branch.
3. Verify width of compare matches the intended value width.

Example: minimal “EA audit” instrumentation pattern

A useful technique is to compute the effective address into a temporary register (without changing meaning) and then load via that register. If behavior changes, the addressing mode or base-update semantics were wrong.

```
.text
.global ea_audit_pattern
ea_audit_pattern:
    /* original intent: load from [X0, X1, lsl #3] */
    add     x3, x0, x1, lsl #3   /* compute EA explicitly */
    ldr     x2, [x3]             /* load through explicit EA */
    ret
```

Example: catching a W-write pointer corruption

```
.text
.global detect_w_write_corruption
detect_w_write_corruption:
    /* if X0 is a pointer, this is a red flag */
    mov     w0, w0               /* clears upper 32 bits of X0 */
    ldr     x1, [x0]             /* may fault or read wrong data */
    ret
```

Appendices

Appendix A — Minimal Register Reference

General-Purpose Registers Summary

AArch64 provides 31 general-purpose integer registers. Each register has a 64-bit view (`Xn`) and a 32-bit view (`Wn`).

Register	Width	Architectural Meaning
X0–X30	64-bit	General-purpose integer registers
W0–W30	32-bit	Low 32-bit view of X0–X30
X30 (LR)	64-bit	Link register (return address)

Core rules:

- `Wn` aliases the low 32 bits of `Xn`.
- Writing to `Wn` clears the upper 32 bits of `Xn`.
- There is no independent 32-bit register file.

Special Registers Summary

Register	Width	Purpose
SP	64-bit	Stack pointer (restricted usage)
PC	64-bit	Program counter (control-flow state)
XZR	64-bit	Zero register (reads as zero, writes discarded)
WZR	32-bit	Zero register (32-bit view)
PSTATE	—	Processor state (flags and control bits)

Discipline:

- `SP` must remain properly aligned and is not a scratch register.
- `PC` is not directly writable; control flow changes it.
- Writes to `XZR/WZR` are discarded.

PSTATE Bit Overview

Only a subset of PSTATE is visible and relevant for core programming.

Bit	Name	Meaning
N	Negative	Result sign bit (signed interpretation)
Z	Zero	Result is zero
C	Carry	Unsigned carry / no borrow
V	Overflow	Signed overflow

Notes:

- NZCV are updated only by flag-setting instructions.
- Flag computation depends on operand width (W vs X).
- Other PSTATE fields are managed by system software.

Register Access Rules (Quick Lookup)

Rule	Meaning
Write to Wn	Clears upper 32 bits of Xn
Pointers use Xn	Prevents address truncation
SP for stack only	Not interchangeable with Xn
Use XZR/WZR	Guaranteed zero value
PC via branches	No arithmetic writes to PC
Flags persist	Until overwritten

Canonical Zeroing Patterns

```
.text
.global zero_examples
zero_examples:
    mov     x0, xzr         /* clear 64-bit register */
    mov     w1, wzr         /* clear 32-bit register */
    ret
```

Canonical Pointer Discipline

```
.text
.global pointer_rules
pointer_rules:
    add     x0, x0, #16     /* correct pointer arithmetic */
    /* add  w0, w0, #16 */  /* WRONG: truncates pointer */
    ret
```

Canonical Flag Usage Discipline

```
.text
.global flag_rules
flag_rules:
    cmp     x0, #0          /* defines NZCV */
    b.eq    is_zero
    mov     w0, #1
    ret
is_zero:
    mov     w0, #0
    ret
```

Canonical Stack Discipline

```
.text
.global stack_rules
stack_rules:
    stp     x19, x20, [sp, #-16]!   /* push */
    /* ... */
    ldp     x19, x20, [sp], #16     /* pop */
    ret
```

Mental Checklist (Before Debugging)

- Did any `W`-write touch a value later used as `X`?
- Are all addresses computed in `X` registers?
- Are flags set immediately before being consumed?
- Is `SP` only adjusted by aligned constants?
- Is each addressing mode intentional and correct?

Appendix B — Instruction Categories Covered

This appendix summarizes the instruction categories referenced in Booklet 12 at a **core architectural level**. The focus is strictly on instruction groups that interact with: register state, memory access, addressing modes, and PSTATE (NZCV).

Data Movement Instructions
Conceptual scope:

- Register-to-register data transfer at selected width.
- Immediate constant construction using wide-move sequences.
- Explicit zero-extension and sign-extension between widths.
- PC-relative address materialization for position-independent code.

Representative mnemonics:

- `MOV`
- `MOVZ`, `MOVK`, `MOVN`
- `UXTW`, `SXTW`, `SXTB`, `SXTH`
- `ADR`, `ADRP`

```
.text
.global data_movement_examples
data_movement_examples:
    mov     x0, x1              /* 64-bit register copy */
    mov     w2, w3              /* 32-bit copy, clears upper X2 */
```

```
    movz    x4, #0x1122, lsl #48
    movk    x4, #0x3344, lsl #32
    movk    x4, #0x5566, lsl #16
    movk    x4, #0x7788, lsl #0

    uxtw    x5, w0              /* zero-extend 32->64 */
    sxtw    x6, w0              /* sign-extend 32->64 */

    adrp    x7, some_symbol
    add     x7, x7, :lo12:some_symbol
    ret

.data
.align 3
some_symbol:
    .quad   0
```

Load/Store Instructions

Conceptual scope:

- Size-specific memory access (byte, halfword, word, doubleword).
- Explicit signed vs unsigned load semantics.
- Paired load/store for stack and block operations.
- Immediate, register, pre-index, and post-index addressing.

Representative mnemonics:

- `LDR`, `STR`

- `LDRB`, `STRB`, `LDRH`, `STRH`
- `LDRSB`, `LDRSH`, `LDRSW`
- `LDP`, `STP`
- `LDUR`, `STUR` (unscaled forms)

```
.text
.global load_store_examples
load_store_examples:
    mov     w1, #0xAA
    strb    w1, [x0, #0]

    mov     w1, #0x1234
    strh    w1, [x0, #2]

    mov     w1, #0x89ABCDEF
    str     w1, [x0, #4]

    mov     x1, #0x1122334455667788
    str     x1, [x0, #8]

    ldrb    w2, [x0, #0]
    ldrsb   x3, [x0, #0]

    ldr     w4, [x0, #4]
    ldrsw   x5, [x0, #4]

    stp     x19, x20, [sp, #-16]!
    ldp     x19, x20, [sp], #16
```

```
    ldr     w6, [x0], #4
    str     w6, [x0, #-4]!
    ret
```

Flag-Setting Instructions

Conceptual scope:

- Compare and test operations producing NZCV only.
- Arithmetic and logical operations that update NZCV.
- Conditional execution driven by PSTATE flags.

Representative mnemonics:

- `CMP`, `CMN`, `TST`
- `ADDS`, `SUBS`, `ANDS`
- `B.{cond}`, `CSET`, `CSEL`

```
.text
.global flag_examples
flag_examples:
    cmp     w0, #0
    b.eq    is_zero

    adds    w1, w0, #1
    cset    w2, mi

    tst     w0, #(1 << 3)
```

```
    cset    w3, ne

    subs    x4, x5, x6
    csel    x0, x5, x6, hs
    ret
is_zero:
    mov     w0, #0
    ret
```

Addressing-Relevant Instruction Groups

Conceptual scope:

- Effective address computation in registers.
- Base update via pre-index and post-index modes.
- Scaled register indexing for element access.
- Control-flow and PC-relative address patterns.

Representative mnemonics:

- `ADD`, `SUB` (with shifted registers)
- `ADR`, `ADRP`
- `B`, `B.{cond}`, `BL`, `BR`, `RET`
- `LDR`/`STR` with base update addressing

```
.text
.global addressing_group_examples
addressing_group_examples:
```

```
    add     x3, x0, x1, lsl #3

    ldr     x2, [x3]

    ldr     w4, [x0], #4
    str     w4, [x0, #-4]!

    stp     x29, x30, [sp, #-16]!
    ldp     x29, x30, [sp], #16

    b       label
    mov     w0, #0
label:
    ret
```

Appendix C — Preparation for Next Booklets

This appendix defines the exact architectural readiness required before progressing to subsequent AArch64 booklets in the CPU Programming Series. It connects the concepts covered in this booklet to the next layers: calling conventions, privilege, memory ordering, and ABI rules.

Readiness for Stack and Calling Conventions
Before studying AArch64 calling conventions and ABI rules, the reader must be fully comfortable with the following architectural facts:

- `SP` is a restricted register used only for stack management.
- Stack growth direction and alignment are architectural constraints.
- Base-update addressing (pre/post-index) is the foundation of stack frames.
- `LR (X30)` holds return addresses but is not automatically preserved.

Architectural expectation: The reader must be able to read and reason about stack manipulation without relying on ABI naming conventions.

```
.text
.global stack_readiness_example
stack_readiness_example:
    stp     x29, x30, [sp, #-16]!   /* allocate stack frame */
    mov     x29, sp                 /* frame pointer (conceptual) */
    /* function body */
    ldp     x29, x30, [sp], #16     /* deallocate stack frame */
    ret
```

Readiness for Exception Levels and Privilege

This booklet intentionally avoids exception levels and system state, but prepares the reader for them conceptually.

Required understanding:

- General-purpose registers are shared across exception levels.
- `PSTATE` contains both user-visible flags and privileged control bits.
- Control flow is always explicit; privilege does not change ISA semantics.

Architectural expectation: The reader must clearly distinguish between: *instruction behavior* and *execution privilege*.

```
.text
.global privilege_neutral_example
privilege_neutral_example:
    cmp     x0, #0              /* flag logic independent of
     ↪  privilege */
    b.eq    zero_case
    mov     w0, #1
    ret
zero_case:
    mov     w0, #0
    ret
```

Readiness for Memory Ordering and Atomics

Before introducing atomics and memory ordering rules, the reader must already understand what the ISA does *not* guarantee.

Required understanding:

- Ordinary loads and stores do not imply ordering across cores.
- Addressing modes define *where* memory is accessed, not *when*.
- Instruction order is not the same as memory visibility order.

Architectural expectation: The reader must treat ordinary `LDR/STR` as non-atomic and non-ordering unless explicitly constrained by later instructions.

```
.text
.global non_atomic_example
non_atomic_example:
    ldr     x1, [x0]        /* no ordering or atomicity guarantee */
    str     x1, [x0, #8]
    ret
```

Mapping Concepts to ABI-Specific Rules

This booklet establishes *architecture-first* reasoning. ABI rules are layered on top of these concepts, not replacements for them.

The reader must already understand:

- Which registers are architecturally general-purpose.
- That register preservation is an ABI decision, not an ISA rule.
- That stack layout conventions are ABI-defined.

Architectural expectation: When reading ABI documentation, the reader must be able to separate:

What the hardware requires vs **what the ABI mandates**.

```
.text
.global abi_independent_example
abi_independent_example:
    /* architecture allows using any X register */
    mov     x5, x0
    add     x5, x5, #8
    ret
```

Completion Checklist Before Proceeding

The reader should confidently answer *yes* to all of the following:

- Can I track register width effects (W vs X) without error?
- Can I compute effective addresses mentally for all addressing modes?
- Do I understand exactly when flags are set and consumed?
- Do I know what the ISA guarantees — and what it explicitly does not?
- Can I read stack and pointer-manipulating code without ABI hints?

If any answer is uncertain, the corresponding chapter in this booklet should be reviewed before continuing to the next volume.

References

ARM Architecture Reference Manuals (Conceptual Use)

This booklet is grounded in the architectural model defined by the ARMv8-A / AArch64 architecture reference manuals. These manuals are used *conceptually*, not as verbatim specifications, to establish:

- precise register semantics (X/W aliasing, zero register behavior),
- PSTATE flag definition and update rules,
- addressing modes and effective address formation,
- architectural guarantees versus system-defined behavior.

Discipline applied in this booklet:

- Only ISA-level guarantees are assumed.
- No ABI, OS, or microarchitectural behavior is inferred unless architecturally defined.
- All examples respect architectural constraints without relying on implementation quirks.

```
.text
.global arch_reference_example
```

```
arch_reference_example:
    /* pure architectural behavior: register move and compare */
    mov     x0, x1
    cmp     x0, #0
    b.eq    zero_case
    ret
zero_case:
    mov     w0, wzr
    ret
```

ISA Documentation and Instruction Semantics

Instruction semantics in this booklet follow the authoritative ISA definitions for:

- operand width selection (W vs X forms),
- flag-setting versus non-flag-setting variants,
- signed versus unsigned loads and extensions,
- base update timing for pre-index and post-index addressing.

Each instruction example is written to make its architectural effect explicit and auditable by inspection.

Semantic emphasis used throughout:

- instruction width is always intentional,
- extension behavior is never implicit,
- base register modification is always visible in the syntax.

```
.text
.global isa_semantics_example
isa_semantics_example:
    ldr     w1, [x0]        /* zero-extends into X1 */
    ldrsw   x2, [x0]        /* sign-extends into X2 */
    adds    x3, x1, x2      /* sets NZCV */
    ret
```

Compiler-Generated Code Observations

Although this booklet does not depend on any compiler behavior, practical validation was performed by examining compiler-generated AArch64 code to confirm alignment with architectural rules.
Observed patterns used as validation (not as authority):

- consistent use of X registers for pointers,
- systematic W writes to enforce zero-extension,
- paired load/store for stack save/restore,
- post-index addressing for pointer iteration.

Important boundary: Compiler output is treated as an *illustration* of correct ISA usage, never as a substitute for architectural specification.

```
.text
.global compiler_style_observation
compiler_style_observation:
    stp     x29, x30, [sp, #-16]!
```

```
    mov     x29, sp
    ldr     w0, [x0]
    add     x0, x0, #4
    ldp     x29, x30, [sp], #16
    ret
```

Cross-References to Other Booklets in This Series

This booklet intentionally isolates the **AArch64 core architectural layer**. Subsequent booklets build directly on these foundations.

Direct conceptual dependencies:

- Stack manipulation and base-update addressing → calling conventions.
- PSTATE flags and condition codes → control flow and ABI rules.
- Addressing modes and alignment → memory ordering and atomics.
- Register width rules → interoperability with higher-level languages.

Series progression discipline:

- Architecture first, ABI second.
- Instruction semantics before optimization.
- Correctness before performance.

```
.text
.global series_transition_example
series_transition_example:
```

```
    /* architectural foundation for later ABI discussion */
    stp     x19, x20, [sp, #-16]!
    /* ABI rules decide whether x19/x20 must be preserved */
    ldp     x19, x20, [sp], #16
    ret
```