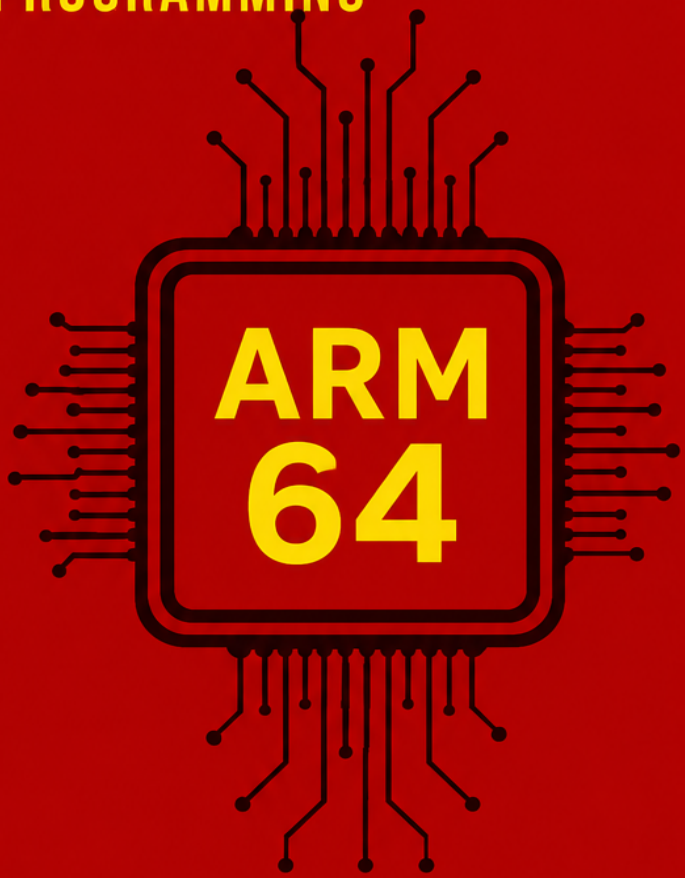
AARCH64 ASSEMBLY PROGRAMMING

FROM FUNDAMENTALS TO ADVANCED SYSTEMS PROGRAMMING

- ARM64 ARCHITECTURE & INSTRUCTION SETS
- SIMD/NEON AND SVE VECTOR PROGRAMMING
- CRYPTOGRAPHIC EXTENSIONS
- MEMORY, CACHE, AND PERFORMANCE
- CONCURRENCY AND SYNCHRONIZATION
- DEBUGGING AND REVERSE ENGINEERING
- COMPILER INTERACTIONS AND INLINE ASSEMBLY
- SYSTEMS PROGRAMMING ON ARM64



JEAN MOREAU

AArch64 (ARM64) Assembly Programming

From Fundamentals to Advanced Systems Programming

Steve T. Publications

This book is available at
https://leanpub.com/aarch64arm64assemblyprogramming

This version was published on 2026-07-15

Contents

From Fundamentals to Advanced Systems Programming

A comprehensive guide to ARM64 architecture, instruction sets, and low-level development. This book takes you from the basics of the AArch64 register file and instruction encoding through the full breadth of the instruction set, including SIMD/NEON vector programming, cryptographic extensions, and SVE scalable vectors. You will learn calling conventions and the AAPCS64 ABI, memory management and cache hierarchy, synchronization primitives for concurrent programming, performance tuning techniques, compiler interactions with inline assembly, debugging and reverse engineering workflows, and practical systems programming on ARM64 platforms. Whether you are optimizing hot paths, writing kernel code, or exploring the architecture for the first time, this book provides the depth and practical examples you need.

First Edition, 2026

This work is provided as a technical reference and educational resource. While every effort has been made to ensure accuracy, processor microarchitectural details vary between implementations, and instruction timings are subject to change across silicon revisions. Readers should consult the relevant Technical Reference Manual (TRM) for their specific target processor when writing performance-critical code.

Introduction: The ARM Revolution and Why AArch64 Matters

From Acorn to Apple Silicon: The ARM Story

In 1983, a small team of twelve engineers at Acorn Computers in Cambridge, England, began working on something that would quietly reshape the entire computing industry. Frustrated by the cost and complexity of licensing Motorola's 68000 processor, they set out to design their own. The result was the ARM1, the first Acorn RISC Machine, which debuted in the Acorn Archimedes personal computer in 1987 at a time when competing home computers were still using 8-bit processors [34].

The architecture's defining characteristic was simplicity. While Intel and Motorola were building increasingly complex CISC (Complex Instruction Set Computer) processors with variable-length instructions and intricate addressing modes, ARM embraced the RISC (Reduced Instruction Set Computing) philosophy that was gaining traction in academic circles: simple, fixed-length instructions that execute in a single cycle, a generous register file, and a strict load/store architecture where only dedicated load and store instructions access memory [2].

In 1990, Acorn, Apple Computer, and VLSI Technology formed a joint venture called Advanced RISC Machines Ltd., renaming the architecture from "Acorn RISC Machine" to "Advanced RISC Machines." This was the birth of ARM as an independent company. The model ARM pioneered, licensing chip designs rather than manufacturing silicon itself, would become the foundation of its extraordinary success [35].

For decades, ARM processors powered the background of computing: embedded systems, set-top boxes, and eventually mobile devices. The turning point came in 2007 with the iPhone, which used an ARM processor to deliver unprecedented performance per watt. What followed was the smartphone revolution, and ARM became the de facto standard for mobile computing, powering virtually every smartphone, tablet, and wearable device on the planet.

By 2011, ARM released the ARMv8-A architecture, introducing AArch64, the 64-bit execution state that is the subject of this book. Apple's Cyclone core was the first consumer implementation [1]. The transition to 64 bits doubled the register width, added more registers, and opened the door to addressing memory spaces far larger than the 4 GB limit of 32-bit systems.

The real surprise came in 2020, when Apple announced the M1 chip for Mac computers, marking the third major architecture transition in Mac history (after Motorola 68000 to PowerPC, and PowerPC to Intel). The M1, built on ARM architecture, delivered performance that rivaled or surpassed competing x86 processors while consuming a fraction of the power. Amazon's Graviton series, AWS's custom ARM-based server chips, demonstrated similar results in the cloud [10]. ARM had arrived not just as the architecture of choice for low-power devices, but as a serious contender for high-performance computing.

Today, ARM processors are everywhere: smartphones, tablets, laptops, servers, embedded controllers, automotive systems, and increasingly in AI accelerators and networking equipment. As of 2025, ARM-based server instances represent a significant and growing share of cloud computing workloads. AWS alone had deployed Graviton-based instances across more than 60 EC2 instance families, and over 17 percent of global data centers incorporated ARM servers compared to just 9 percent in 2022 [10]. The Android ecosystem alone runs on billions of ARM devices worldwide, making AArch64 arguably the most widely deployed 64-bit processor architecture in history.

Why Learn Assembly in 2026?

In an era where compilers are extraordinarily sophisticated and high-level languages dominate software development, the question is natural: why learn assembly language at all? The answer is not that you should write your entire application in assembly. It is that understanding what happens beneath the compiler's abstraction layer fundamentally changes how you write code, debug problems, and reason about performance.

Consider these concrete scenarios where AArch64 assembly knowledge matters:

Performance-critical code paths. When profiling reveals that a particular function consumes 40 percent of your application's runtime, the optimizer may have done its best with the source code you gave it. Understanding

the generated assembly lets you restructure the algorithm, provide better hints through intrinsics or inline assembly, or identify missed optimization opportunities like vectorization or instruction-level parallelism.

Systems programming and kernel development. Operating system kernels, device drivers, bootloaders, and firmware all require assembly for their lowest layers. The context switch, interrupt handler entry and exit, page fault resolution, and MMU initialization are written in assembly because they operate at a level where no higher runtime exists yet.

Security research and reverse engineering. Understanding AArch64 binary code is essential for vulnerability analysis, malware detection, exploit development, and understanding how compiled programs behave. With ARM's dominance in mobile, the ability to read and analyze ARM64 binaries has become as important as x86-64 disassembly for security professionals.

Embedded systems and bare-metal programming. Many embedded applications run without an operating system, directly on the hardware. Initialization code, startup routines, and low-level hardware control are written in assembly, and understanding the architecture is non-negotiable for this work.

Compiler development and optimization. If you work on compilers, language runtimes, or JIT engines, you need to understand the target architecture's capabilities, register allocation strategies, instruction scheduling, and calling conventions. AArch64's clean RISC design makes it an excellent teaching platform for compiler internals.

Interfacing with existing libraries. Many high-performance libraries expose assembly-optimized routines for critical operations: cryptographic primitives, SIMD-accelerated math, and memory management functions. Understanding these interfaces at the assembly level helps you use them correctly and diagnose problems when things go wrong.

What This Book Covers and How to Use It

This book is structured as a progressive journey through the AArch64 architecture, from foundational concepts to advanced topics. Each chapter builds on the previous ones, but individual chapters are designed to be substantially self-contained so you can reference specific topics without reading everything in sequence.

The early chapters establish the fundamentals: the register file, instruction encoding, execution states, and the basic instruction categories (data processing, load/store, branching). These chapters include extensive code examples with thorough comments explaining each instruction's effect.

The middle chapters dive deeper into specific domains: floating-point and SIMD/NEON vector programming, the calling convention and ABI, exception levels and privilege modes, memory management and the cache hierarchy, synchronization and concurrent programming, and cryptographic extensions. Each of these chapters contains real-world code examples drawn from actual systems programming scenarios.

The later chapters focus on practical engineering: performance tuning and optimization techniques, working with compilers through inline assembly and intrinsics, debugging workflows, reverse engineering considerations, and interfaces between user space and the operating system kernel.

Throughout the book, you will find comparisons with x86-64 where they illuminate AArch64's design choices. If you have experience with x86 assembly, these comparisons provide anchors for understanding the differences. If you are new to assembly entirely, the comparisons help contextualize ARM's place in the broader computing landscape.

Code examples use standard GNU assembler syntax and are tested against the toolchains available on mainstream Linux distributions. Most examples can be assembled and linked with `as` and `ld`, or compiled as part of C programs using GCC or Clang inline assembly. Where platform-specific behavior matters (such as differences between Linux, macOS on Apple Silicon, and bare-metal execution), these differences are called out explicitly.

The book assumes basic programming knowledge (familiarity with variables, functions, memory, and pointers from any mainstream language) but no prior assembly experience. If you can write a loop in C, you can follow along. The goal is that by the end, you will read AArch64 assembly fluently, understand how the hardware executes it, and know how to write your own when the situation demands it.

Setting Up Your Development Environment

Before writing assembly code, you need a working toolchain. The following options cover the most common scenarios:

Native development on ARM64 Linux. If you are running Linux on an ARM64 device (Raspberry Pi 4/5, AWS Graviton instance, or Apple Silicon Mac with Linux via UTM/Asahi), the GNU assembler (`as`) and linker (`ld`) are typically available through the `binutils` package. GCC and Clang both support AArch64 natively:

```
# Assemble and link a standalone program
as hello.s -o hello.o
ld hello.o -o hello
./hello

# Compile C with assembly mixed in
gcc -O2 -march=armv8-a main.c my_func.s -o program
```

Cross-compilation from x86-64 to ARM64. When developing on an x86 workstation for ARM64 targets, install a cross-compilation toolchain:

```
# Debian/Ubuntu
sudo apt install gcc-aarch64-linux-gnu g++-aarch64-linux-gnu

# Use the cross-compiler
aarch64-linux-gnu-gcc -O2 -march=armv8-a program.c -o program
aarch64-linux-gnu-as hello.s -o hello.o
```

QEMU for emulation and testing. QEMU's user-mode emulation allows running AArch64 binaries on x86 hosts without native hardware:

```
# Run an AArch64 binary under QEMU
qemu-aarch64 ./aarch64_program

# Cross-compile and run in one step
aarch64-linux-gnu-gcc -o test program.c
qemu-aarch64 ./test
```

Compiler Explorer (godbolt.org). For quick experimentation and learning, the online Compiler Explorer service supports AArch64 GCC and Clang compilers. Enter C/C++ code, select the ARM64 target, and see the generated assembly update in real time. This is invaluable for understanding how specific C constructs map to AArch64 instructions.

Chapter 1: Architecture Fundamentals

The AArch64 architecture represents a deliberate departure from the decades of architectural baggage that accumulated in x86. Where x86-64 grew incrementally from the 16-bit 8086 through 32-bit x86 to 64-bit AMD64, AArch64 was designed from the ground up as a clean 64-bit RISC architecture. This chapter establishes the foundational concepts you need before writing any instruction: the register file, execution states, how instructions are encoded in binary, and the design choices that distinguish AArch64 from its competitors.

The Register File: Thirty-One General-Purpose Registers

At the heart of AArch64's design is a generous, flat register file. The architecture provides thirty-one general-purpose registers, each 64 bits wide, named X0 through X30. This is nearly four times as many as x86-64's eight 64-bit general-purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, R8-R15), and the difference has profound implications for code density, compiler optimization quality, and how often values must be spilled to memory [5]. The large register pool is one of AArch64's most consequential design choices: it reduces memory traffic, improves instruction-level parallelism by keeping more values live in registers, and simplifies the compiler's register allocation problem.

```
+--------+--------+--------+--------+
|   X0   |   X1   |   X2   |   X3   |
+--------+--------+--------+--------+
|   X4   |   X5   |   X6   |   X7   |
+--------+--------+--------+--------+
|   X8   |   X9   |  X10   |  X11   |
+--------+--------+--------+--------+
|  X12   |  X13   |  X14   |  X15   |
+--------+--------+--------+--------+
|  X16   |  X17   |  X18   |  X19   |
+--------+--------+--------+--------+
|  X20   |  X21   |  X22   |  X23   |
+--------+--------+--------+--------+
|  X24   |  X25   |  X26   |  X27   |
+--------+--------+--------+--------+
|  X28   |  X29   |  X30   |  XZR   |
+--------+--------+--------+--------+
```

Each register can be accessed at multiple granularities. The full 64-bit register is named Xn (for example, X5). The lower 32 bits are accessible as Wn (W5). Writing to a W register zero-extends the value into the full 64-bit register, which is an important detail for maintaining correct state:

```
1       mov w0, #0x1234      // X0 = 0x0000000000001234 (upper 32 bits zeroed)
2       mov x0, #0xDEADBEEF  // X0 = 0x00000000DEADBEEF
```

This zero-extension behavior eliminates the partial-register stalls that plague x86, where writing to AL (the lowest 8 bits of RAX) requires the processor to merge the result back into the full register. On AArch64, writing to Wn always produces a clean, predictable result.

Special registers. Register 31 has a dual role depending on context. When referenced as XZR or WZR, it is a hardwired zero register: reading it always returns zero, and writes to it are silently discarded. This eliminates the need for a separate "move-immediate-zero" instruction:

```
1       add x0, x1, xzr      // X0 = X1 + 0, effectively MOV X0, X1
2       and w0, w1, wzr      // W0 = W1 & 0, effectively MOV W0, #0
```

When register 31 is used as the stack pointer (SP), it provides the current stack address. The choice between XZR and SP for register 31 is determined by the instruction encoding itself, not by any mode switch.

Named roles within the ABI. While all thirty-one registers are architecturally identical, the Application Binary Interface assigns conventional roles to specific registers:

- **X0-X7**: Parameter passing and return values. The first eight integer/-pointer arguments to a function are passed in these registers, and the return value is placed in X0. These are caller-saved (volatile): a called function may clobber them without restoring the original values.
- **X8**: Used as the indirect result pointer for large struct returns, and as the syscall number register when making kernel system calls. Also caller-saved.
- **X9-X15**: General-purpose caller-saved registers available for temporary computations.
- **X16-X17**: Intraprocess call-saved registers (IP0, IP1). Used by the linker and for interprocedural scratch space.

- **X18**: Platform register (the platform register reservation is implementation-defined; on some platforms it is reserved for the operating system).
- **X19–X28**: Callee-saved (non-volatile) registers. A function that modifies any of these must save the original value on the stack and restore it before returning.
- **X29**: Frame pointer (FP). Points to the current stack frame's base, enabling stack unwinding and debugging.
- **X30**: Link register (LR). Holds the return address for subroutine calls. The BL (branch with link) instruction stores the return address here automatically.

The Program Counter. Unlike x86, where RIP is a general-purpose register that can be freely read and used in arithmetic, AArch64's program counter is not directly accessible as a general-purpose register. You cannot write `MOV X0, PC` to read the current instruction address. Instead, the ADRP and ADR instructions provide position-independent addressing relative to the current PC, which is the mechanism compilers use for accessing local data and computing function addresses:

```
1       adrp x0, local_data   // X0 = page-aligned address of local_data
2       add  x0, x0, :lo12:local_data  // X0 += offset within the page
```

This design reflects a broader AArch64 philosophy: rather than providing general-purpose access to architectural state, the architecture provides specific instructions that do exactly what you need in the common case, without the flexibility to misuse them.

Execution States and Modes: AArch64 vs AArch32

ARMv8-A defines two execution states, each with its own register width and instruction set. Understanding the distinction between them is essential for understanding what code can run on a given processor and how privilege transitions work [6].

AArch64 state. This is the 64-bit execution state, unique to ARMv8 and later architectures. In AArch64 state:

- General-purpose registers are 64 bits wide
- The program counter, stack pointer, and exception link registers are all 64 bits
- Only the A64 instruction set can be executed (fixed 32-bit instructions)
- Privilege is organized into four exception levels (EL0 through EL3)

AArch32 state. This is the 32-bit execution state, maintained for backward compatibility with ARMv7-A and earlier code. In AArch32 state:

- General-purpose registers are 32 bits wide (R0–R15, plus banked SPsr and SPSR)
- The A32 (ARM, 32-bit fixed-width) and T32 (Thumb, 16/32-bit mixed-width) instruction sets can be executed
- Privilege uses the ARMv7 model with user, FIQ, IRQ, supervisor, abort, and system modes

A critical constraint governs state transitions: **the execution state can only change at exception level boundaries.** You cannot switch from AArch64 to AArch32 (or vice versa) within a single exception level. A 64-bit operating system running at EL1 in AArch64 state can host 32-bit applications that run at EL0 in AArch32 state, but the reverse is impossible: a 32-bit OS cannot run 64-bit applications because the higher exception level must use AArch64 for the lower level to use it.

```
EL3 (Secure Monitor)    : AArch64 only (hardware-fixed)
EL2 (Hypervisor)        : AArch64 or AArch32 (configurable)
EL1 (OS Kernel)         : AArch64 or AArch32 (configurable by EL2/EL3)
EL0 (User Applications) : AArch64 or AArch32 (configurable by EL1)
```

For the vast majority of this book, we work exclusively in AArch64 state. The AArch32 discussion is included here for completeness and to explain compatibility scenarios you may encounter when deploying software across mixed 32-bit and 64-bit environments.

Exception levels as privilege. AArch64 replaces x86's four protection rings (Ring 0 through Ring 3) with a simpler model of exception levels. Only EL0 and EL1 are mandatory; EL2 (hypervisor) and EL3 (secure monitor) are optional but nearly universally implemented in modern processors. Each transition to a higher exception level saves the processor state automatically, including the

return address (in the Exception Link Register, ELR_ELx), the processor state (in SPSR_ELx), and the stack pointer switches to an exception-level-specific stack.

Instruction Encoding and Formats

Every AArch64 instruction is exactly 32 bits wide. This is a fundamental design decision with far-reaching consequences:

- Instructions are always aligned on 4-byte boundaries, simplifying instruction fetch and decode
- The processor never has to determine where one instruction ends and the next begins (unlike x86's variable-length encoding, which can be 1 to 15 bytes)
- Branch targets are naturally 4-byte aligned, freeing the lowest two bits of branch target addresses for metadata
- Code density is lower than Thumb-2 but higher than x86 on average, because the fixed width eliminates prefix bytes and escape sequences

The 32-bit instruction is organized into fields that the decoder reads to determine the operation, operands, and modifiers. While the exact field layout varies by instruction class, several patterns recur:

```
Bits:  31                    24 23        16 15       10 9          5 4     0
      +----------------------+-----------+-----------+------------+-------+
      | Operation-specific   | Rn (base) | Rd (dest) | Rm (source)| ...   |
      | fields / opcode      | reg[4:0]  | reg[4:0]  | reg[4:0]   |       |
      +----------------------+-----------+-----------+------------+-------+
```

Register specifiers are always 5 bits, encoding values 0 through 31. The upper bits of the instruction determine the instruction class. ARM's documentation organizes the decode into a four-level table structure:

1. **Top-level opcode** (bits 31-29 or similar): Determines the broad instruction class (data processing, load/store, branch, system, SIMD/FP, etc.)
2. **Sub-opcode**: Refines within the class (e.g., ADD vs SUB within data processing immediate)
3. **Operand specifier**: Determines operand types and sizes

4. **Register fields**: The actual register operands

Let us look at a concrete example. Consider the instruction `ADD X0, X1, X2` (add X1 and X2, store in X0):

```
Binary: 0101 0100 0100 0001 0010 0000 0000 0000
Hex:    0x8B020000
```

Breaking this down:

- Bits 31-29 = `010`: Data processing (register) class
- Bit 30 = `1`, bit 29 = `0`: 64-bit operation
- Bits 27-24 = `0000`: ADD opcode
- Bits 22-16 encode various modifiers (shift, extend, etc.)
- Bits 15-12 = `0000` = Rd = X0
- Bits 9-5 = `00001` = Rn = X1
- Bits 4-0 = `00010` = Rm = X2

You do not need to memorize these encodings. Understanding the structure helps when reading raw machine code, debugging encoding issues, or building tooling, but day-to-day assembly programming uses mnemonics exclusively. The key takeaway is that the fixed 32-bit width and regular field layout make AArch64 significantly easier to disassemble and analyze than x86.

Immediate value encoding. AArch64 cannot encode arbitrary 64-bit immediates directly in a single instruction (there are only 32 bits available, and some must be used for the opcode and register fields). Instead, immediates use clever encoding tricks:

- **MOVZ** (move with zero): Loads a 16-bit immediate value into a register, zeroing all other bits. The 16-bit value can be placed at any 16-bit boundary within the 64-bit register.
- **MOVK** (move with keep): Replaces a 16-bit chunk of a register with an immediate, preserving all other bits.
- **MOVK/MOVZ sequence**: A full 64-bit constant is loaded using up to four MOVZ/MOVK instructions, one per 16-bit quadlet.

```
    // Load the 32-bit constant 0xDEADBEEF into X0
    movz x0, #0BEEF              // X0 = 0x000000000000BEEF
    movk x0, #0DEAD, LSL #16     // X0 = 0x00000000DEADBEEF

    // Load the 64-bit constant 0x0123456789ABCDEF into X0
    movz x0, #0CDEF              // X0 = 0x000000000000CDEF
    movk x0, #089AB, LSL #16     // X0 = 0x0000000089ABCDEF
    movk x0, #04567, LSL #32     // X0 = 0x00000000456789ABCDEF
    movk x0, #0x0123, LSL #48    // X0 = 0x0000000000123456789ABCDEF
```

For simple small constants, the assembler often generates a single MOV instruction (which is itself an alias for `MOVZ Rd, #imm` with the upper bits zeroed):

```
    mov x0, #42        // Assembled as: movz x0, #42
    mov w0, #-1       // Assembled as: movn w0, #0  (move not: ~0 = 0xFFFFFFFF)
```

The **MVN** (move not) instruction, aliased as `MVN Rd, #imm`, loads the bitwise negation of an immediate. This is useful for creating masks and negative constants efficiently.

Endianness and Byte Ordering

AArch64 supports both little-endian and big-endian byte ordering, though little-endian is by far the dominant choice in practice. Linux on ARM64, macOS on Apple Silicon, and Windows on ARM all use little-endian exclusively. Big-endian support exists primarily for compatibility with networking protocols and legacy embedded systems.

In **little-endian** mode (the default), the least significant byte of a multi-byte value is stored at the lowest memory address:

```
Value 0x41424344 stored at address 0x1000 in little-endian:
Address 0x1000: 0x44  (least significant byte)
Address 0x1001: 0x43
Address 0x1002: 0x42
Address 0x1003: 0x41  (most significant byte)
```

In **big-endian** mode, the order is reversed:

```
Value 0x41424344 stored at address 0x1000 in big-endian:
Address 0x1000: 0x41  (most significant byte)
Address 0x1001: 0x42
Address 0x1002: 0x43
Address 0x1003: 0x44  (least significant byte)
```

The endianness is configured per exception level through the SCTLR_ELx system register's E0E and EEN bits, and can differ between EL0 (user space) and higher exception levels. In practice, all modern AArch64 deployments use little-endian at all exception levels.

Endianness matters most when you:

- Write code that interfaces with network protocols (which use big-endian / "network byte order")
- Read binary data from files or hardware registers with a fixed byte order
- Perform byte-level manipulations on multi-byte values
- Debug memory dumps and need to interpret raw bytes correctly

The REV, REV16, REV32, and REVSH instructions provide efficient byte-reversal operations for converting between endiannesses:

```
    rev  x0, x1      // Reverse byte order of 64-bit value
    rev16 w0, w1     // Reverse byte order within each 16-bit halfword
    rev32 w0, w1     // Reverse byte order within 32-bit word (useful for
    ↪  network conversions)
```

Comparison with x86-64 Architecture Design

Understanding AArch64 is often easier if you already know x86-64. The two architectures share the same fundamental goal, to execute programs efficiently on general-purpose processors, but they arrived at very different solutions through decades of divergent evolution.

Register count and naming. AArch64's thirty-one general-purpose registers dwarf x86-64's eight. This means fewer memory accesses for temporary values, better compiler register allocation, and assembly code that is often easier to follow because values stay in registers longer. The trade-off is that each register specifier requires 5 bits (log2(31) rounded up) versus 4 bits for x86-64's 16 registers (including the 8 original plus 8 added in 64-bit mode).

Instruction encoding. x86-64 instructions vary from 1 to 15 bytes, with prefixes, ModR/M bytes, SIB bytes, and displacement/imm fields that must be decoded sequentially. AArch64's fixed 32-bit width means the decoder always knows exactly where each field is, enabling parallel decode of multiple instructions per cycle. This is a significant advantage for deep out-of-order pipelines.

Memory access. AArch64 is a strict load/store architecture: only LDR and STR instructions (and their variants) can access memory. All arithmetic, logical, and data manipulation operates exclusively on registers. x86-64 allows most arithmetic instructions to operate directly on memory operands (`ADD [RAX], RCX`), which can produce shorter code but complicates the execution pipeline because the ALU must wait for a memory read to complete before it can compute.

Conditional execution. x86-64 has conditional moves (CMOVcc) and always executes branches with some form of prediction. AArch64 inherited from its 32-bit predecessor the ability to make any data processing instruction conditional based on the condition flags, though this feature was significantly reduced in AArch64 compared to ARMv7. Modern AArch64 relies primarily on conditional branches rather than widespread conditional execution, but conditional selects (CSEL) and compare-and-branch (CBZ/CBNZ) provide efficient alternatives to branching for simple cases.

Stack alignment. The AAPCS64 ABI requires the stack pointer to be 16-byte aligned at all public calling points. x86-64's System V ABI has the same requirement, while the Windows x64 ABI requires 16-byte alignment after the CALL instruction (meaning the caller must ensure 8-byte alignment before CALL, since CALL pushes an 8-byte return address).

Privilege model. x86-64's four protection rings are mostly reduced to two in practice (Ring 0 for kernel, Ring 3 for user), with Rings 1 and 2 unused. AArch64's exception level model is cleaner: EL0 is always user space, EL1 is the OS kernel (or hypervisor if EL2 is not implemented), EL2 is the hypervisor when virtualization is used, and EL3 is reserved for secure monitor code (TrustZone). The hierarchy is strict: you can only trap to the immediately higher level.

Atomic operations. x86-64 has the LOCK prefix that turns many instructions into atomic operations on a cache-coherent bus (`LOCK XADD`, `LOCK CMPXCHG`). AArch64 uses an explicit load-exclusive/store-exclusive (LDXR/STXR) pattern, which is more flexible (it works correctly even across non-cache-

coherent systems with appropriate barriers) but requires more instructions to implement common atomic patterns. ARMv8.1 added Large System Extension (LSE) atomics that provide single-instruction compare-and-swap operations for better scalability on large systems.

These differences are not merely academic. They shape how compilers generate code, how you write assembly, and what optimization opportunities exist. The AArch64 design favors regularity and predictability, which benefits both hardware implementers and software developers who need to reason about performance characteristics.

Chapter 2: Addressing Modes and Data Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

The Load/Store Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Immediate Offsets and Pre-/Post-Indexed Addressing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Register-Offset and Shifted Addressing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Unaligned Access and Endianness Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

The AArch64 Memory Model: Weak Ordering Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Chapter 3: Integer Operations and the ALU

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Data Processing: Immediate Instructions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Arithmetic: ADD, SUB, and Their Variants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Multiplication and Division

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Logical Operations: AND, ORR, EOR, BIC

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Shifts, Rotates, and Bit Manipulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Comparison and Set Operations

Chapter 4: Control Flow and Branching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Unconditional and Conditional Branches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

The Condition Flag System: CZFO vs x86 Flags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Compare and Branch (CBZ/CBNZ)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Loop Patterns and Idiom Recognition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Branch Prediction and Speculative Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Tail Call Optimization and Return Sequences

Chapter 5: Floating-Point and SIMD/NEON

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

The FP/SIMD Register File: V0 to V31

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Scalar Floating-Point Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

NEON Vector Data Types and Layouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Vector Arithmetic and Reduction Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Data Conversion and Type Casting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Real-World SIMD Optimization Case Study

Chapter 6: Calling Conventions and the AAPCS64 ABI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

The AAPCS64 Specification Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Register Allocation: Arguments, Temporaries, Preserved Registers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Stack Frame Layout and Alignment Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Parameter Passing: Integers, Floats, and Structs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Variadic Functions and Varargs Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Interoperation with C, Rust, and Other Languages

Chapter 7: Exception Levels and Privilege Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Four Exception Levels: EL0 through EL3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Entering and Returning from Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

System Registers: Configuration and Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Interrupt Handling and Vector Tables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Context Switching at the Assembly Level

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Secure vs Non-Secure World

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Chapter 8: Memory Management and Cache Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Virtual Memory and Translation Regimes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

The MMU and Page Table Walks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

TLB Management and Maintenance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Cache Hierarchy: L1, L2, L3, and Data Caches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Cache Maintenance Instructions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Memory Attributes and Access Permissions

Chapter 9: Synchronization and Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Atomic Instructions: LDAXR, STLR, CAS Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Load-Acquire and Store-Release Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Memory Barriers: DMB, DSB, ISB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Lock-Free Programming Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Spinlocks and Mutex Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Chapter 10: Cryptographic Extensions and Advanced Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

ARMv8 Cryptographic Instructions: AES, PMULL, SHA

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Polynomial Multiplication and GCM Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

SHA-256 Hardware Acceleration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

SVE: Scalable Vector Extension Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

SVE2: Matrix and Bitmanip Extensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Other Extensions: RDM, LSE, DP, MTE

Chapter 11: Performance Tuning and Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Pipeline Architecture and Instruction-Level Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Out-of-Order Execution and Scoreboarding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Branch Prediction Patterns and Misprediction Costs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Cache-Friendly Code Layout and Data Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Profiling with perf and Hardware Counters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Real-World Optimization Case Study

Chapter 12: Compiler Interactions and Inline Assembly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

How Compilers Generate AArch64 Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

GCC Inline Assembly Syntax and Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Clang/LLVM Inline Assembly and Intrinsics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Optimization Barriers and Volatile Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Reading Compiler Output with objdump and llvm-objdump

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

When to Use Assembly vs Intrinsics vs Compiler Hints

Chapter 13: Debugging, Reverse Engineering, and Systems Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Debugging with GDB, QEMU, and Hardware Debuggers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Reverse Engineering AArch64 Binaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Common Obfuscation Patterns and Anti-Analysis Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Syscall Interface: The ARM64 System Call ABI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Kernel Entry Points and Trap Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Practical Systems Programming Examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Conclusion: The Future of AArch64 and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

What You Have Learned: Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

ARMv9 and the Next Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

The Server Revolution Continues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Where to Go From Here

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Glossary of ARM-Specific Terms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Appendix A: AArch64 Instruction Categories Quick Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Appendix B: Linux ARM64 System Call Table (Selected)

Appendix C: Memory Barrier Semantics Quick Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Appendix D: AAPCS64 Register Roles Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

General-Purpose Registers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

FP/SIMD Registers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Stack Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.

Argument Passing Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/aarch64arm64assemblyprogramming.