ARM64 (AArch64) Assembly Reference

For Bare-Metal & OS Development on ARMv8-A

Based on ARM Architecture Reference Manual

1 Registers

1.1 General Purpose Registers

Register	64-bit / 32-bit	Purpose
`x0-x7` / `w0-w7`	Arguments & return values	Function parameters (AAPCS64)
`x8` / `w8`	Indirect result location	Struct return pointer
`x9-x15` / `w9-w15`	Temporary (caller-saved)	Scratch registers
`x16-x17` / `w16-w17`	Intra-procedure-call	Linker veneers (IP0, IP1)
`x18` / `w18`	Platform register	Reserved (don't use)
`x19-x28` / `w19-w28`	Callee-saved	Must preserve across calls
`x29` / `w29`	Frame pointer (FP)	Stack frame base
`x30` / `w30`	Link register (LR)	Return address
`sp`	Stack pointer	Must be 16-byte aligned
`xzr` / `wzr`	Zero register	Reads as 0, writes discarded
`pc`	Program counter	Not directly accessible

Note: `w` registers are the lower 32 bits of `x` registers. Writing to `w` zeros upper 32 bits.

1.2 Special System Registers (EL1)

Register	Purpose
`MPIDR_EL1`	Multiprocessor Affinity Register (core ID)
`SCTLR_EL1`	System Control Register (MMU, caches enable)
`VBAR_EL1`	Vector Base Address Register (exception table)
`ELR_EL1`	Exception Link Register (return address)
`SPSR_EL1`	Saved Program Status Register
`ESR_EL1`	Exception Syndrome Register (exception cause)
`FAR_EL1`	Fault Address Register
`SP_EL0`	Stack Pointer for EL0
`CurrentEL`	Current Exception Level

2 Data Movement Instructions

2.1 MOV — Move

Syntax	Description
`mov Xd, Xn`	Copy register: `Xd = Xn`
`mov Xd, #imm`	Load immediate (limited to 16 bits or bitmask)
`mov Xd, sp`	Copy stack pointer to register
`mov sp, Xn`	Set stack pointer from register

Limitation: Cannot load arbitrary 64-bit values. Use `movz`/`movk` or `ldr`.

2.2 MOVZ / MOVK / MOVN — Build Large Immediates

Syntax	Description
`movz Xd, #imm16, lsl #N`	Zero register, insert 16-bit imm at position N
`movk Xd, #imm16, lsl #N`	Keep other bits, insert 16-bit imm at position N
`movn Xd, #imm16, lsl #N`	Move NOT (inverted immediate)

Shift positions: N = 0, 16, 32, or 48

Example — Load 0xFE201000:

```
movz x0, #0x1000            // x0 = 0x0000000000001000
movk x0, #0xFE20, lsl #16   // x0 = 0x00000000FE201000
```

2.3 LDR / STR — Load and Store

Syntax	Description
`ldr Xt, [Xn]`	Load 64-bit from address in Xn
`ldr Wt, [Xn]`	Load 32-bit from address in Xn
`ldrb Wt, [Xn]`	Load byte (8-bit), zero-extend
`ldrh Wt, [Xn]`	Load halfword (16-bit), zero-extend
`ldrsb Xt, [Xn]`	Load byte, sign-extend to 64-bit
`ldrsh Xt, [Xn]`	Load halfword, sign-extend to 64-bit
`ldrsw Xt, [Xn]`	Load word (32-bit), sign-extend to 64-bit
`str Xt, [Xn]`	Store 64-bit to address in Xn
`str Wt, [Xn]`	Store 32-bit to address in Xn
`strb Wt, [Xn]`	Store byte (lower 8 bits)
`strh Wt, [Xn]`	Store halfword (lower 16 bits)

2.4 Addressing Modes

Mode	Syntax	Meaning
Base	`[Xn]`	addr = Xn
Offset	`[Xn, #imm]`	addr = Xn + imm
Pre-index	`[Xn, #imm]!`	Xn += imm, addr = Xn
Post-index	`[Xn], #imm`	addr = Xn, Xn += imm
Register offset	`[Xn, Xm]`	addr = Xn + Xm
Scaled	`[Xn, Xm, lsl #3]`	addr = Xn + (Xm ¡¡ 3)

Pre-index example: `str x0, [sp, #-16]!` → Push (decrement sp, then store)

Post-index example: `ldr x0, [sp], #16` → Pop (load, then increment sp)

2.5 LDP / STP — Load/Store Pair

Syntax	Description
`ldp Xt1, Xt2, [Xn]`	Load two 64-bit values
`ldp Xt1, Xt2, [Xn, #imm]`	Load pair with offset
`ldp Xt1, Xt2, [Xn], #imm`	Load pair, post-index
`ldp Xt1, Xt2, [Xn, #imm]!`	Load pair, pre-index
`stp Xt1, Xt2, [Xn]`	Store two 64-bit values

Common pattern — Save/restore registers:

```
stp x29, x30, [sp, #-16]!    // Push FP and LR
...
ldp x29, x30, [sp], #16      // Pop FP and LR
```

2.6 ADR / ADRP — PC-Relative Address

Syntax	Description
`adr Xd, label`	Xd = PC + offset to label (±1MB range)
`adrp Xd, label`	Xd = 4KB-aligned page of label (±4GB range)

ADRP pattern — Load symbol address:

```
adrp x0, my_var              // x0 = page containing my_var
add  x0, x0, :lo12:my_var    // x0 += offset within page
```

2.7 LDR (Literal) — PC-Relative Load

Syntax	Description
`ldr Xt, =value`	Pseudo-instruction: load from literal pool
`ldr Xt, label`	Load from PC-relative address

3 Arithmetic Instructions

3.1 ADD / SUB — Addition and Subtraction

Syntax	Description
`add Xd, Xn, Xm`	Xd = Xn + Xm
`add Xd, Xn, #imm`	Xd = Xn + imm (12-bit immediate)
`add Xd, Xn, Xm, lsl #N`	Xd = Xn + (Xm ¡¡ N)
`adds Xd, Xn, Xm`	Add and set flags (NZCV)
`sub Xd, Xn, Xm`	Xd = Xn - Xm
`sub Xd, Xn, #imm`	Xd = Xn - imm
`subs Xd, Xn, Xm`	Subtract and set flags
`neg Xd, Xm`	Xd = 0 - Xm (negate)

3.2 MUL / DIV — Multiplication and Division

Syntax	Description
`mul Xd, Xn, Xm`	Xd = Xn * Xm (lower 64 bits)
`smull Xd, Wn, Wm`	Signed multiply 32×32→64
`umull Xd, Wn, Wm`	Unsigned multiply 32×32→64
`sdiv Xd, Xn, Xm`	Signed divide: Xd = Xn / Xm
`udiv Xd, Xn, Xm`	Unsigned divide: Xd = Xn / Xm
`msub Xd, Xn, Xm, Xa`	Xd = Xa - (Xn * Xm) (modulo helper)

Modulo pattern: No direct MOD instruction. Use: `msub Xd, Xn, Xm, Xa` where Xa = original dividend.

```
udiv x2, x0, x1      // x2 = x0 / x1
msub x3, x2, x1, x0  // x3 = x0 - (x2 * x1) = x0 % x1
```

4 Logical & Bit Instructions

4.1 AND / ORR / EOR / BIC — Bitwise Operations

Syntax	Description
`and Xd, Xn, Xm`	Xd = Xn & Xm (bitwise AND)
`and Xd, Xn, #imm`	Xd = Xn & imm (bitmask immediate)
`ands Xd, Xn, Xm`	AND and set flags
`orr Xd, Xn, Xm`	Xd = Xn — Xm (bitwise OR)
`eor Xd, Xn, Xm`	Xd = Xn ˆ Xm (bitwise XOR)
`bic Xd, Xn, Xm`	Xd = Xn & ∼Xm (bit clear)
`mvn Xd, Xm`	Xd = ∼Xm (bitwise NOT)
`orn Xd, Xn, Xm`	Xd = Xn — ∼Xm (OR NOT)
`eon Xd, Xn, Xm`	Xd = Xn ˆ ∼Xm (XOR NOT)

Bitmask immediates: Not arbitrary values! Must be repeating patterns. Assembler will error if invalid.

4.2 Shift Operations

Syntax	Description
`lsl Xd, Xn, #imm`	Logical shift left by immediate
`lsl Xd, Xn, Xm`	Logical shift left by register
`lsr Xd, Xn, #imm`	Logical shift right (zero fill)
`asr Xd, Xn, #imm`	Arithmetic shift right (sign extend)
`ror Xd, Xn, #imm`	Rotate right

4.3 Bit Field Instructions

Syntax	Description
`ubfx Xd, Xn, #lsb, #w`	Extract w bits from position lsb, zero-extend
`sbfx Xd, Xn, #lsb, #w`	Extract w bits, sign-extend
`bfi Xd, Xn, #lsb, #w`	Insert w bits at position lsb
`bfc Xd, #lsb, #w`	Clear w bits at position lsb

4.4 Other Bit Operations

Syntax	Description
`clz Xd, Xn`	Count leading zeros
`cls Xd, Xn`	Count leading sign bits
`rbit Xd, Xn`	Reverse bits
`rev Xd, Xn`	Reverse bytes (endianness swap)
`rev16 Xd, Xn`	Reverse bytes in each halfword
`rev32 Xd, Xn`	Reverse bytes in each word

5 Comparison & Condition Flags

5.1 CMP / CMN / TST

Syntax	Description
`cmp Xn, Xm`	Compare: set flags from Xn - Xm
`cmp Xn, #imm`	Compare with immediate
`cmn Xn, Xm`	Compare negative: flags from Xn + Xm
`tst Xn, Xm`	Test bits: flags from Xn & Xm
`tst Xn, #imm`	Test bits with immediate

5.2 Condition Flags (NZCV)

Flag	Name	Set When
N	Negative	Result bit 63 (or 31) is 1
Z	Zero	Result is zero
C	Carry	Unsigned overflow (add) or no borrow (sub)
V	Overflow	Signed overflow

5.3 Condition Codes

Code	Meaning	Flags	Use Case
`eq`	Equal	Z=1	a == b
`ne`	Not equal	Z=0	a != b
`cs/hs`	Carry set / unsigned ≥	C=1	a ¿= b (unsigned)
`cc/lo`	Carry clear / unsigned <	C=0	a ¡ b (unsigned)
`mi`	Minus (negative)	N=1	result ¡ 0
`pl`	Plus (positive or zero)	N=0	result ¿= 0
`vs`	Overflow set	V=1	signed overflow
`vc`	Overflow clear	V=0	no overflow
`hi`	Unsigned higher	C=1 & Z=0	a ¿ b (unsigned)
`ls`	Unsigned lower or same	C=0 — Z=1	a ¡= b (unsigned)
`ge`	Signed ≥	N=V	a ¿= b (signed)
`lt`	Signed <	N!=V	a ¡ b (signed)
`gt`	Signed >	Z=0 & N=V	a ¿ b (signed)
`le`	Signed ≤	Z=1 — N!=V	a ¡= b (signed)
`al`	Always	—	unconditional

6 Branch Instructions

6.1 Unconditional Branches

Syntax	Description
`b label`	Branch to label (±128MB range)
`bl label`	Branch with link (call): LR = return addr
`br Xn`	Branch to address in register
`blr Xn`	Branch with link to register
`ret`	Return (branch to LR / x30)
`ret Xn`	Return to address in Xn

6.2 Conditional Branches

Syntax	Description
`b.cond label`	Branch if condition true (±1MB)
`cbz Xn, label`	Branch if Xn == 0 (±1MB)
`cbnz Xn, label`	Branch if Xn != 0
`tbz Xn, #bit, label`	Branch if bit #bit of Xn is 0 (±32KB)
`tbnz Xn, #bit, label`	Branch if bit #bit of Xn is 1

Examples:

```
cmp  x0, x1
b.eq equal_label       // Branch if x0 == x1
b.lt less_than         // Branch if x0 < x1 (signed)
b.lo less_unsigned     // Branch if x0 < x1 (unsigned)

cbz  x0, is_zero       // Branch if x0 == 0
tbz  x0, #31, positive // Branch if bit 31 is clear (positive)
```

7 Conditional Operations

7.1 CSEL / CSET / CINC — Conditional Select

Syntax	Description
`csel Xd, Xn, Xm, cond`	Xd = cond ? Xn : Xm
`csinc Xd, Xn, Xm, cond`	Xd = cond ? Xn : Xm+1
`csinv Xd, Xn, Xm, cond`	Xd = cond ? Xn : ∼Xm
`csneg Xd, Xn, Xm, cond`	Xd = cond ? Xn : -Xm
`cset Xd, cond`	Xd = cond ? 1 : 0
`csetm Xd, cond`	Xd = cond ? -1 : 0
`cinc Xd, Xn, cond`	Xd = cond ? Xn+1 : Xn

Example — branchless absolute value:

```
cmp  x0, #0
csneg x0, x0, x0, ge   // if (x0 >= 0) x0 = x0 else x0 = -x0
```

8 System Instructions

8.1 MRS / MSR — System Register Access

Syntax	Description
`mrs Xd, sysreg`	Move from system register to Xd
`msr sysreg, Xn`	Move from Xn to system register

Examples:

```
mrs x0, MPIDR_EL1     // Get core ID
mrs x0, CurrentEL     // Get current exception level
msr VBAR_EL1, x0      // Set exception vector base
msr DAIFSet, #0xf     // Disable all interrupts
msr DAIFClr, #0xf     // Enable all interrupts
```

8.2 Exception & Interrupt Instructions

Syntax	Description
`svc #imm`	Supervisor call (syscall to EL1)
`hvc #imm`	Hypervisor call (to EL2)
`smc #imm`	Secure monitor call (to EL3)
`eret`	Exception return
`wfi`	Wait for interrupt (sleep until interrupt)
`wfe`	Wait for event
`sev`	Send event

8.3 Memory Barriers

Syntax	Description
`dmb ish`	Data Memory Barrier — ensure memory accesses complete
`dsb ish`	Data Synchronization Barrier — wait for all memory ops
`isb`	Instruction Synchronization Barrier — flush pipeline

When needed:

- After writing to system registers (use `isb`)
- Before/after MMIO operations (use `dsb`)
- When ordering matters for device registers (use `dmb`)

8.4 Cache Operations

Syntax	Description
`dc ivac, Xn`	Invalidate data cache by address
`dc cvac, Xn`	Clean data cache by address
`dc civac, Xn`	Clean and invalidate by address
`ic iallu`	Invalidate all instruction caches

9 Assembler Directives

Directive	Purpose
`.section .text`	Code section
`.section .data`	Initialized data section
`.section .bss`	Uninitialized data section
`.globl symbol`	Export symbol (make visible to linker)
`.extern symbol`	Declare external symbol
`.align N`	Align to 2^N bytes
`.balign N`	Align to N bytes
`.byte val`	Emit 1 byte
`.hword val`	Emit 2 bytes (halfword)
`.word val`	Emit 4 bytes
`.quad val`	Emit 8 bytes
`.ascii "str"`	String without null terminator
`.asciz "str"`	Null-terminated string
`.space N`	Reserve N bytes (zeros in .bss)
`.equ name, val`	Define constant

10 Common Patterns for OS Development

10.1 Check Core ID (Only Boot Core 0)

```
    mrs  x0, MPIDR_EL1       // Read multiprocessor affinity register
    and  x0, x0, #3          // Mask to get core ID (bits 0-1)
    cbz  x0, primary_core    // If core 0, continue
    b    hang                // Other cores: infinite loop
```

10.2 Set Stack Pointer

```
    ldr  x0, =_stack_top     // Load stack top address
    mov  sp, x0              // Set stack pointer
```

10.3 Clear BSS

```
    adr  x0, __bss_start     // Start of BSS
    adr  x1, __bss_end       // End of BSS
clear_loop:
    cmp  x0, x1
    b.ge done
    str  xzr, [x0], #8       // Store 0, post-increment
    b    clear_loop
done:
```

10.4 Function Prologue/Epilogue

```
my_func:
    stp  x29, x30, [sp, #-16]!   // Save FP and LR
    mov  x29, sp                  // Set frame pointer
    // ... function body ...
    ldp  x29, x30, [sp], #16     // Restore FP and LR
```

```
    ret
```

10.5 Busy-Wait Loop (UART Ready)

```
wait_ready:
    ldr  w1, [x0, #UART_FR]  // Load flag register
    tst  w1, #UART_TXFF      // Test TX FIFO full bit
    b.ne wait_ready          // If full, keep waiting
    str  w2, [x0, #UART_DR]  // Write character
```

11 Quick Reference Card

Data Movement	Arithmetic	Logic/Bit	Branch
`mov Xd, Xn`	`add Xd, Xn, Xm`	`and Xd, Xn, Xm`	`b label`
`movz Xd, #imm`	`sub Xd, Xn, Xm`	`orr Xd, Xn, Xm`	`bl label`
`movk Xd, #imm`	`mul Xd, Xn, Xm`	`eor Xd, Xn, Xm`	`ret`
`ldr Xd, [Xn]`	`udiv Xd, Xn, Xm`	`bic Xd, Xn, Xm`	`b.cond label`
`str Xd, [Xn]`	`sdiv Xd, Xn, Xm`	`lsl Xd, Xn, #n`	`cbz Xn, label`
`ldp X1, X2, [Xn]`	`cmp Xn, Xm`	`lsr Xd, Xn, #n`	`cbnz Xn, label`
`stp X1, X2, [Xn]`	`adds Xd, Xn, Xm`	`tst Xn, #imm`	`tbz Xn, #b, lbl`
`adr Xd, label`	`subs Xd, Xn, Xm`	`mvn Xd, Xm`	`tbnz Xn, #b, lbl`
System	**Conditional**	**Memory**	**Exception**
`mrs Xd, sysreg`	`csel Xd,Xn,Xm,c`	`dmb ish`	`svc #imm`
`msr sysreg, Xn`	`cset Xd, cond`	`dsb ish`	`eret`
`nop`	`cinc Xd, Xn, c`	`isb`	`wfi`

ARM Documentation: `https://developer.arm.com/documentation/ddi0487/latest` (Architecture Reference Manual)
Programmer's Guide: `https://developer.arm.com/documentation/den0024/latest`