ARM Instruction Set Reference

ARMv8-A / AArch64 with FP, SIMD, System Extensions + Classic AArch32

ARM Operands

Name	Example	Comments
31 GP registers	`X0`–`X30` / `W0`–`W30`	64-bit (`Xn`) or 32-bit (`Wn`) views. `X30` is the link register (LR). `XZR/WZR` is the zero register.
Stack pointer	`SP`	Dedicated stack pointer; accessible by a limited set of instructions. Not a general-purpose register.
Program counter	`PC`	Not directly accessible as a GP register; read via `ADR/ADRP`.
32 FP/SIMD regs	`V0`–`V31`	128-bit SIMD/FP registers. Viewed as `Bn` (8b), `Hn` (16b), `Sn` (32b), `Dn` (64b), or `Qn` (128b).
Condition flags	`NZCV`	Negative, Zero, Carry, oVerflow — set by comparison and flag-setting instructions.

ARM Assembly Language

Category	Instruction	Example	Meaning	Comments	ISA
Arithmetic	Add	`ADD X5, X6, X7`	`X5 = X6 + X7`	Three register operands; add	**AArch64**
Arithmetic	Add immediate	`ADD X5, X6, #20`	`X5 = X6 + 20`	Add 12-bit immediate (opt. shift 12)	**AArch64**
Arithmetic	Add and set flags	`ADDS X5, X6, X7`	`X5 = X6 + X7`	Sets NZCV condition flags	**AArch64**
Arithmetic	Subtract	`SUB X5, X6, X7`	`X5 = X6 - X7`	Three register operands; subtract	**AArch64**
Arithmetic	Subtract imm.	`SUB X5, X6, #20`	`X5 = X6 - 20`	Subtract 12-bit immediate	**AArch64**
Arithmetic	Sub and set flags	`SUBS X5, X6, X7`	`X5 = X6 - X7`	Sets NZCV; used for comparisons	**AArch64**
Arithmetic	Negate	`NEG X5, X6`	`X5 = 0 - X6`	Pseudo: `SUB X5, XZR, X6`	**AArch64**
Arithmetic	Add with carry	`ADC X5, X6, X7`	`X5 = X6 + X7 + C`	Multi-word addition	**AArch64**
Compare	Compare	`CMP X5, X6`	`NZCV from X5 - X6`	Alias for `SUBS XZR, X5, X6`	**AArch64**
Compare	Compare imm.	`CMP X5, #20`	`NZCV from X5 - 20`	Sets flags; result discarded	**AArch64**
Compare	Compare negative	`CMN X5, X6`	`NZCV from X5 + X6`	Alias for `ADDS XZR, X5, X6`	**AArch64**
Compare	Test bits	`TST X5, X6`	`NZCV from X5 & X6`	Alias for `ANDS XZR, X5, X6`	**AArch64**
Data Transfer	Load register	`LDR X5, [X6, #40]`	`X5=Memory[X6+40]`	Load 64-bit from memory	**AArch64**
Data Transfer	Load 32-bit	`LDR W5, [X6, #40]`	`W5=Memory[X6+40]`	Load 32-bit; zero-extends to 64	**AArch64**
Data Transfer	Load signed byte	`LDRSB X5, [X6]`	`X5=SignExt(Mem[X6])`	Sign-extend byte to 64 bits	**AArch64**
Data Transfer	Load halfword	`LDRH W5, [X6]`	`W5=Memory[X6]`	Load 16-bit; zero-extend	**AArch64**
Data Transfer	Load signed half	`LDRSH X5, [X6]`	`X5=SignExt(Mem[X6])`	Sign-extend halfword to 64 bits	**AArch64**
Data Transfer	Store register	`STR X5, [X6, #40]`	`Memory[X6+40]=X5`	Store 64-bit to memory	**AArch64**
Data Transfer	Store 32-bit	`STR W5, [X6]`	`Memory[X6]=W5`	Store lower 32 bits	**AArch64**
Data Transfer	Store byte	`STRB W5, [X6]`	`Memory[X6]=W5[7:0]`	Store least significant byte	**AArch64**

Category	Instruction	Example	Meaning	Comments	ISA
Data Transfer	Store halfword	`STRH W5, [X6]`	`Memory[X6]=W5[15:0]`	Store least significant halfword	**AArch64**
Data Transfer	Load pair	`LDP X5,X6,[X7]`	`X5,X6=Mem[X7]`	Load two registers; efficient push/pop	**AArch64**
Data Transfer	Store pair	`STP X5,X6,[X7]`	`Mem[X7]=X5,X6`	Store two registers at once	**AArch64**
Data Transfer	Pre-index	`LDR X5, [X6, #8]!`	`X6+=8; X5=Mem[X6]`	Update base before access	**AArch64**
Data Transfer	Post-index	`LDR X5, [X6], #8`	`X5=Mem[X6]; X6+=8`	Update base after access	**AArch64**
Data Transfer	PC-relative addr	`ADR X5, label`	`X5 = PC + offset`	±1 MB PC-relative address	**AArch64**
Data Transfer	PC-rel page addr	`ADRP X5, label`	`X5 = PC[page]+off`	±4 GB page-aligned address	**AArch64**
Move	Move immediate	`MOV X5, #0xFF`	`X5 = 0xFF`	Load 16-bit imm (with optional shift)	**AArch64**
Move	Move register	`MOV X5, X6`	`X5 = X6`	Alias for `ORR X5, XZR, X6`	**AArch64**
Move	Move wide imm.	`MOVZ X5, #0x1234, LSL#16`	`X5 = 0x12340000`	Move 16-bit imm into shifted position	**AArch64**
Move	Move keep	`MOVK X5, #0xAB`	`X5[15:0] = 0xAB`	Insert 16-bit imm; keep other bits	**AArch64**
Move	Move NOT	`MVN X5, X6`	`X5 = ~X6`	Bitwise NOT of register	**AArch64**
Logical	And	`AND X5, X6, X7`	`X5 = X6 & X7`	Bitwise AND	**AArch64**
Logical	And set flags	`ANDS X5, X6, X7`	`X5 = X6 & X7`	AND and set NZCV flags	**AArch64**
Logical	Inclusive or	`ORR X5, X6, X7`	`X5 = X6 | X7`	Bitwise OR	**AArch64**
Logical	Or NOT	`ORN X5, X6, X7`	`X5 = X6 | ~X7`	OR with inverted second operand	**AArch64**
Logical	Exclusive or	`EOR X5, X6, X7`	`X5 = X6 ^ X7`	Bitwise XOR	**AArch64**
Logical	Bit clear	`BIC X5, X6, X7`	`X5 = X6 & ~X7`	AND with inverted second operand	**AArch64**
Shift	Logical shift left	`LSL X5, X6, #3`	`X5 = X6 << 3`	Shift left by immediate	**AArch64**
Shift	Logical shift right	`LSR X5, X6, #3`	`X5 = X6 >> 3`	Logical shift right (zero fill)	**AArch64**
Shift	Arith shift right	`ASR X5, X6, #3`	`X5 = X6 >> 3`	Arithmetic shift right (sign fill)	**AArch64**
Shift	Rotate right	`ROR X5, X6, #3`	`X5 = RotR(X6,3)`	Rotate right by immediate	**AArch64**
Shift	Shift left (reg)	`LSL X5, X6, X7`	`X5 = X6 << X7`	Alias for `LSLV`; shift by register	**AArch64**
Mult/Div.	Multiply	`MUL X5, X6, X7`	`X5 = X6 * X7`	Lower 64 bits of product	**AArch64**
Mult/Div.	Multiply-accumulate	`MADD X5,X6,X7,X8`	`X5 = X8 + X6*X7`	Fused multiply-add	**AArch64**
Mult/Div.	Multiply-subtract	`MSUB X5,X6,X7,X8`	`X5 = X8 - X6*X7`	Fused multiply-subtract	**AArch64**
Mult/Div.	Signed mult high	`SMULH X5, X6, X7`	`X5 = (X6*X7)[127:64]`	Upper 64 bits, signed×signed	**AArch64**
Mult/Div.	Unsigned mult high	`UMULH X5, X6, X7`	`X5 = (X6*X7)[127:64]`	Upper 64 bits, unsigned×unsigned	**AArch64**
Mult/Div.	Signed divide	`SDIV X5, X6, X7`	`X5 = X6 / X7`	Signed integer division	**AArch64**
Mult/Div.	Unsigned divide	`UDIV X5, X6, X7`	`X5 = X6 / X7`	Unsigned integer division	**AArch64**
Bit Manip.	Count leading zeros	`CLZ X5, X6`	`X5 = clz(X6)`	Count leading zero bits	**AArch64**
Bit Manip.	Reverse bytes	`REV X5, X6`	`X5 = bswap(X6)`	Byte-reverse (endian swap)	**AArch64**
Bit Manip.	Bit field extract	`UBFX X5,X6,#4,#8`	`X5 = X6[11:4]`	Extract bit field, zero-extend	**AArch64**
Bit Manip.	Bit field insert	`BFI X5,X6,#4,#8`	`X5[11:4] = X6[7:0]`	Insert bit field into register	**AArch64**
Bit Manip.	Sign extend byte	`SXTB X5, W6`	`X5 = SignExt(W6[7:0])`	Sign-extend byte to 64 bits	**AArch64**
Bit Manip.	Zero extend half	`UXTH W5, W6`	`W5 = W6[15:0]`	Zero-extend halfword	**AArch64**

Category	Instruction	Example	Meaning	Comments	ISA
Cond. Branch	Branch if equal	`B.EQ label`	`if(Z==1) PC=label`	Branch if last comparison was equal	**AArch64**
Cond. Branch	Branch if not equal	`B.NE label`	`if(Z==0) PC=label`	Branch if not equal	**AArch64**
Cond. Branch	Branch if less (s)	`B.LT label`	`if(N!=V) PC=label`	Signed less than	**AArch64**
Cond. Branch	Branch if $\geq$ (s)	`B.GE label`	`if(N==V) PC=label`	Signed greater than or equal	**AArch64**
Cond. Branch	Branch if less (u)	`B.LO label`	`if(C==0) PC=label`	Unsigned lower (carry clear)	**AArch64**
Cond. Branch	Branch if $\geq$ (u)	`B.HS label`	`if(C==1) PC=label`	Unsigned higher or same	**AArch64**
Cond. Branch	Compare & branch = 0	`CBZ X5, label`	`if(X5==0) PC=label`	Compare and branch if zero	**AArch64**
Cond. Branch	Compare & branch $\neq$ 0	`CBNZ X5, label`	`if(X5!=0) PC=label`	Compare and branch if non-zero	**AArch64**
Cond. Branch	Test & branch = 0	`TBZ X5,#3,label`	`if(X5[3]==0) PC=label`	Test single bit and branch	**AArch64**
Uncond. Branch	Branch	`B label`	`PC = label`	$\pm$128 MB PC-relative jump	**AArch64**
Uncond. Branch	Branch with link	`BL label`	`X30=PC+4; PC=label`	Function call; saves return addr in LR	**AArch64**
Uncond. Branch	Branch to register	`BR X5`	`PC = X5`	Indirect branch	**AArch64**
Uncond. Branch	Branch link reg.	`BLR X5`	`X30=PC+4; PC=X5`	Indirect call via register	**AArch64**
Uncond. Branch	Return	`RET`	`PC = X30`	Alias for `BR X30`; function return	**AArch64**
Cond. Select	Cond select	`CSEL X5,X6,X7,EQ`	`X5=(EQ)?X6:X7`	Select between two registers on cond	**AArch64**
Cond. Select	Cond increment	`CSINC X5,X6,X7,NE`	`X5=(NE)?X6:X7+1`	Select or increment	**AArch64**
Cond. Select	Cond set	`CSET X5, EQ`	`X5=(EQ)?1:0`	Set register to 0 or 1 on condition	**AArch64**
Floating Point	FP load	`LDR S5, [X6, #40]`	`S5=Memory[X6+40]`	Load 32-bit float from memory	*FP*
Floating Point	FP store	`STR D5, [X6]`	`Memory[X6]=D5`	Store 64-bit double to memory	*FP*
Floating Point	FP add	`FADD S5, S6, S7`	`S5 = S6 + S7`	Single-precision FP add	*FP*
Floating Point	FP subtract	`FSUB D5, D6, D7`	`D5 = D6 - D7`	Double-precision FP subtract	*FP*
Floating Point	FP multiply	`FMUL S5, S6, S7`	`S5 = S6 * S7`	Single-precision FP multiply	*FP*
Floating Point	FP divide	`FDIV D5, D6, D7`	`D5 = D6 / D7`	Double-precision FP divide	*FP*
Floating Point	FP square root	`FSQRT S5, S6`	`S5 = sqrt(S6)`	Single-precision square root	*FP*
Floating Point	FP fused multiply-add	`FMADD D5,D6,D7,D8`	`D5 = D8 + D6*D7`	Fused multiply-add (no rounding loss)	*FP*
Floating Point	FP compare	`FCMP S5, S6`	`NZCV from S5 - S6`	Sets NZCV flags; result discarded	*FP*
Floating Point	FP int$\rightarrow$float	`SCVTF S5, W6`	`S5 = (float)W6`	Signed int to single-precision	*FP*

Category	Instruction	Example	Meaning	Comments	ISA
Floating Point	FP float→int	`FCVTZS W5, S6`	`W5 = (int)S6`	Single-precision to signed int (trunc)	*FP*
Floating Point	FP move	`FMOV D5, D6`	`D5 = D6`	Copy FP register	*FP*
SIMD (Neon)	Vector add	`ADD V0.4S,V1.4S,V2.4S`	4×32-bit add	Element-wise addition	*SIMD*
SIMD (Neon)	Vector multiply	`MUL V0.4S,V1.4S,V2.4S`	4×32-bit mul	Element-wise multiply	*SIMD*
SIMD (Neon)	Vector FP add	`FADD V0.4S,V1.4S,V2.4S`	4×FP32 add	SIMD single-precision FP add	*SIMD*
SIMD (Neon)	Vector FP fused MA	`FMLA V0.4S,V1.4S,V2.4S`	$V0 += V1 * V2$	Fused multiply-accumulate per lane	*SIMD*
SIMD (Neon)	Load multiple	`LD1 {V0.4S},[X1]`	Load 4×32b to V0	Contiguous SIMD load	*SIMD*
SIMD (Neon)	Store multiple	`ST1 {V0.4S},[X1]`	Store V0 to memory	Contiguous SIMD store	*SIMD*
SIMD (Neon)	Duplicate scalar	`DUP V0.4S, W5`	Broadcast W5 to all lanes	Splat integer to vector	*SIMD*
Atomic (LSE)	Atomic add	`LDADD X5, X6, [X7]`	`X6=Mem[X7]; Mem[X7]+=X5`	Load-add atomic (LSE extension)	**AArch64**
Atomic (LSE)	Atomic swap	`SWP X5, X6, [X7]`	`X6=Mem[X7]; Mem[X7]=X5`	Atomic swap in memory	**AArch64**
Atomic (LSE)	Atomic CAS	`CAS X5, X6, [X7]`	`if(Mem[X7]==X5) Mem=X6`	Compare-and-swap (LSE)	**AArch64**
Atomic (LSE)	Load-acquire	`LDAR X5, [X6]`	`X5=Mem[X6]`	Load with acquire semantics	**AArch64**
Atomic (LSE)	Store-release	`STLR X5, [X6]`	`Mem[X6]=X5`	Store with release semantics	**AArch64**
Atomic (LSE)	Load-excl	`LDXR X5, [X6]`	`X5=Mem[X6]`	Exclusive load; 1st half of LL/SC	**AArch64**
Atomic (LSE)	Store-excl	`STXR W7,X5,[X6]`	`Mem[X6]=X5; W7=0/1`	Exclusive store; 2nd half of LL/SC	**AArch64**
Memory Order	Data mem barrier	`DMB ISH`	—	Ensure ordering of data accesses	*System*
Memory Order	Data sync barrier	`DSB ISH`	—	Ensure completion of data accesses	*System*
Memory Order	Instr sync barrier	`ISB`	—	Flush pipeline; sync instr stream	*System*
System	System register read	`MRS X5, NZCV`	`X5 = NZCV`	Move system register to GP register	*System*
System	System register write	`MSR NZCV, X5`	`NZCV = X5`	Move GP register to system register	*System*
System	Supervisor call	`SVC #0`	—	System call (trap to EL1)	*System*
System	Hypervisor call	`HVC #0`	—	Trap to EL2 hypervisor	*System*
System	No operation	`NOP`	—	No operation (pipeline hint)	*System*
Arithmetic	Add	`ADD R5, R6, R7`	`R5 = R6 + R7`	Three register operands; add	**AArch32**
Arithmetic	Add with carry	`ADC R5, R6, R7`	`R5 = R6 + R7 + C`	Multi-word addition with carry	**AArch32**
Arithmetic	Subtract	`SUB R5, R6, R7`	`R5 = R6 - R7`	Three register operands; subtract	**AArch32**
Arithmetic	Reverse subtract	`RSB R5, R6, #0`	`R5 = 0 - R6`	Negate; `imm - Rn`	**AArch32**
Arithmetic	Multiply	`MUL R5, R6, R7`	`R5 = R6 * R7`	32-bit result; lower word of product	**AArch32**
Arithmetic	Mult-accumulate	`MLA R5,R6,R7,R8`	`R5 = R8 + R6*R7`	Multiply and add accumulator	**AArch32**

Category	Instruction	Example	Meaning	Comments	ISA
Arithmetic	Signed divide	`SDIV R5, R6, R7`	`R5 = R6 / R7`	Signed division (ARMv7-A+)	**AArch32**
Arithmetic	Unsigned divide	`UDIV R5, R6, R7`	`R5 = R6 / R7`	Unsigned division (ARMv7-A+)	**AArch32**
Compare	Compare	`CMP R5, R6`	`CPSR from R5 - R6`	Sets NZCV flags; result discarded	**AArch32**
Compare	Compare imm.	`CMP R5, #20`	`CPSR from R5 - 20`	Compare register with immediate	**AArch32**
Compare	Test bits	`TST R5, R6`	`CPSR from R5 & R6`	AND and set flags; result discarded	**AArch32**
Data Transfer	Load register	`LDR R5, [R6, #40]`	`R5=Memory[R6+40]`	Load 32-bit word from memory	**AArch32**
Data Transfer	Store register	`STR R5, [R6, #40]`	`Memory[R6+40]=R5`	Store 32-bit word to memory	**AArch32**
Data Transfer	Load byte	`LDRB R5, [R6]`	`R5=Memory[R6]`	Load byte, zero-extend to 32 bits	**AArch32**
Data Transfer	Load halfword	`LDRH R5, [R6]`	`R5=Memory[R6]`	Load 16-bit, zero-extend	**AArch32**
Data Transfer	Load multiple	`LDM R13!, {R0-R3}`	`R0..R3=Mem[R13..]`	Load multiple regs; "!" updates base	**AArch32**
Data Transfer	Load mult. dec.	`LDMDB R5!, {R0-R3}`	`R0..R3=Mem[R5..]`	Decrement-before variant	**AArch32**
Data Transfer	Store multiple	`STM R13!, {R0-R3}`	`Mem[R13..]=R0..R3`	Store multiple regs; "!" updates base	**AArch32**
Data Transfer	Store mult. dec.	`STMDB R5!,{R0-R3}`	`Mem[R5..]=R0..R3`	Decrement-before variant	**AArch32**
Data Transfer	Push	`PUSH {R4-R7, LR}`	`SP-=n; Mem[SP]=regs`	Alias: `STMDB SP!, {...}`	**AArch32**
Data Transfer	Pop	`POP {R4-R7, PC}`	`regs=Mem[SP]; SP+=n`	Alias: `LDM SP!, {...}`	**AArch32**
Data Transfer	Load upper imm.	`MOVW R5, #0x1234`	`R5[15:0]=0x1234`	Move 16-bit imm into lower halfword	**AArch32**
Data Transfer	Move top	`MOVT R5, #0x5678`	`R5[31:16]=0x5678`	Move 16-bit imm into upper halfword	**AArch32**
Move	Move	`MOV R5, R6`	`R5 = R6`	Copy register	**AArch32**
Move	Move NOT	`MVN R5, R6`	`R5 = ~R6`	Bitwise NOT of register	**AArch32**
Move	Move immediate	`MOV R5, #0xFF`	`R5 = 0xFF`	8-bit immediate with rotation	**AArch32**
Logical	And	`AND R5, R6, R7`	`R5 = R6 & R7`	Bitwise AND	**AArch32**
Logical	Inclusive or	`ORR R5, R6, R7`	`R5 = R6 | R7`	Bitwise OR	**AArch32**
Logical	Exclusive or	`EOR R5, R6, R7`	`R5 = R6 ^ R7`	Bitwise XOR	**AArch32**
Logical	Bit clear	`BIC R5, R6, R7`	`R5 = R6 & ~R7`	AND with inverted second operand	**AArch32**
Shift	Logical shift left	`LSL R5, R6, #3`	`R5 = R6 << 3`	Also usable as 2nd-operand shift	**AArch32**
Shift	Logical shift right	`LSR R5, R6, #3`	`R5 = R6 >> 3`	Logical (zero fill) shift right	**AArch32**
Shift	Arith shift right	`ASR R5, R6, #3`	`R5 = R6 >> 3`	Arithmetic (sign fill) shift right	**AArch32**
Shift	Rotate right	`ROR R5, R6, #3`	`R5 = RotR(R6,3)`	Rotate right; also via barrel shifter	**AArch32**
Cond. Branch	Branch if equal	`BEQ label`	`if(Z==1) PC=label`	All ARM32 instrs can be conditional	**AArch32**
Cond. Branch	Branch if not equal	`BNE label`	`if(Z==0) PC=label`	Condition suffix on mnemonic	**AArch32**
Cond. Branch	Branch if less (s)	`BLT label`	`if(N!=V) PC=label`	Signed less than	**AArch32**

Category	Instruction	Example	Meaning	Comments	ISA
Cond. Branch	Branch if $\geq$ (s)	`BGE label`	`if(N==V) PC=label`	Signed greater or equal	**AArch32**
Uncond. Branch	Branch	`B label`	`PC = label`	$\pm$32 MB PC-relative jump	**AArch32**
Uncond. Branch	Branch with link	`BL label`	`LR=PC+4; PC=label`	Subroutine call; return addr in LR	**AArch32**
Uncond. Branch	Branch exchange	`BX LR`	`PC = LR`	Branch and optionally switch to Thumb	**AArch32**
Uncond. Branch	Branch link exch.	`BLX R5`	`LR=PC+4; PC=R5`	Indirect call; can switch to Thumb	**AArch32**
System	Software interrupt	`SVC #0`	—	System call (formerly `SWI`)	**AArch32**
System	Breakpoint	`BKPT #0`	—	Software breakpoint for debugger	**AArch32**
System	No operation	`NOP`	—	No operation	**AArch32**

ISA Key: AArch64 = ARMv8-A 64-bit ISA | **AArch32** = Classic ARM 32-bit ISA (ARMv7-A and earlier) | *FP* = Floating-point | *SIMD* = Advanced SIMD (Neon) | *System* = System, barrier, and exception instructions

AArch64 notes: `Wn` selects the lower 32 bits of `Xn`; writing to `Wn` zeros the upper 32 bits. Suffix `S` (e.g. `ADDS`) sets condition flags. Condition codes: `EQ`, `NE`, `LT`, `LE`, `GT`, `GE` (signed); `LO` (`CC`), `LS`, `HI`, `HS` (`CS`) (unsigned). LSE atomics require ARMv8.1-A or later. SIMD lane arrangements: `.8B`, `.16B`, `.4H`, `.8H`, `.2S`, `.4S`, `.2D`.

AArch32 notes: 16 GP registers `R0`–`R15`: `R13`=SP, `R14`=LR, `R15`=PC. Nearly all instructions support conditional execution via 2-letter suffix (e.g. `ADDEQ`, `MOVNE`). `LDM/STM` addressing modes: `IA` (increment after, default), `IB` (increment before), `DA` (decrement after), `DB` (decrement before). `PUSH` = `STMDB SP!`; `POP` = `LDM SP!`. Barrel shifter allows inline shift of second operand: e.g. `ADD R0, R1, R2, LSL #3`.