

Aarch64 most common instructions

General conventions

rd, rn, rm: w or x registers; op2: register or #immn (n-bit immediate) if {S} is present flags will be affected
Containers: x (64-bit register), w (32-bit register)

| | Instruction | Mnemonic | Syntax | Explanation | Flags |
|----------------------------|--|----------|---------------------------|--|-------|
| Arithmetic operations | Addition | ADD{S} | ADD{S} rd, rn, op2 | rd = rn + op2 | {Yes} |
| | Subtraction | SUB{S} | SUB{S} rd, rn, op2 | rd = rn - op2 | {Yes} |
| | Negation | NEG{S} | NEG{S} rd, op2 | rd = -op2 | {Yes} |
| | with carry | NGC{S} | NGC{S} rd, rm | rd = -rm - ~C | {Yes} |
| | Unsigned multiply | MUL | MUL rd, rn, rm | rd = rn x rm | |
| | Unsigned multiply long | UMULL | UMULL xd, wn, wm | xd = wn x wm | |
| | Unsigned multiply high | UMULH | UMULH xd, xn, xm | xd = <127:64> of xn x xm | |
| | Signed multiply long | SMULL | SMULL xd, xn, xm | xd = wn x wn (signed operands) | |
| | Signed multiply high | SMULH | SMULL xd, wn, wm | xd = <127:64> of xn x xm (signed operands) | |
| | Multiply and add | MADD | MADD rd, rn, rm, ra | rd = ra + (rn x rm) | |
| | Multiply and sub | MSUB | MSUB rd, rn, rm, ra | rd = ra - (rn x rm) | |
| | Multiply and neg | MNEG | MNEG rd, rn, rm | Rd = -(rn x rm) | |
| | Unsigned multiply and add long | UMADDL | UMADDL xd, wn, wm, xa | xd = xa + (wm x wn) | |
| | Unsigned multiply and sub long | UMSUBL | UMSUBL xd, wn, wm, xa | xd = xa - (wm x wn) | |
| | Unsigned multiply and neg long | UMNEGL | UMNEGL xd, wn, wn | Xd = -(wm x wn) | |
| | Signed multiply and add long | SMADDL | SMADDL xd, wn, wm, xa | xd = xa + (wm x wn) | |
| | Signed multiply and sub long | SMSUBL | SMSUBL xd, wn, wm, xa | xd = xa - (wm x wn) | |
| | Signed multiply and neg long | SMNEGL | SMNEGL xd, wn, wm | Xd = - (wm x wn) | |
| | Unsigned divide | UDIV | UDIV rd, rn, rm | rd = rn / rm | |
| | Signed divide | SDIV | SDIV rd, rn, rm | rd = rn / rm | |
| | Note: the remainder may be computed using the MSUB instruction as numerator - (quotient x denominator) | | | | |
| Bitwise logical operations | bitwise AND | AND | AND{S} rd, rn, op2 | rd = rn & op2 | {Yes} |
| | Bitwise AND with neg | BIC | BIC{S} rd, rn, op2 | rd = rn & ~op2 | {Yes} |
| | Bitwise OR | ORR | ORR rd, rn, op2 | rd = rn op2 | |
| | Bitwise OR with neg | ORN | ORN rd, rn, op2 | rd = rn ~op2 | |
| | Bitwise XOR | EOR | EOR rd, rn, op2 | rd = rn @ op2 | |
| | Bitwise XOR with neg | EON | EON rd, rn, op2 | rd = rn @ ~op2 | |
| | Logical shift left | LSL | LSL rd, rn, op2 | Logical shift left (stuffing zeros enter from right) | |
| | Logical shift right | LSR | LSR rd, rn, rm | Logical shift right (stuffing zeros enter from left) | |
| | Arithmetic shift right | ASR | ASR rd, rn, op2 | Arithmetic shift right (preserves sign) | |
| | Rotate right | ROR | ROR rd, rn, op2 | Rotate right (carry not involved) | |
| | Move to register | MOV | MOV rd, op2 | rd = op2 | |
| | Move to register, neg | MVN | MVN rd, op2 | rd = ~op2 | |
| | Test bits | TST | TST rn, op2 | rn & op2 | Yes |
| Bitfield ops | Bitfield insert | BFI | BFI rd, rn, #lsb, #width | Moves a bitfield of #width bits starting at source bit 0 to destination starting at bit #lsb | |
| | Bitfield extract | UBFX | UBFZ rd, rn, #lsb, #width | Moves a bitfield of #width bits starting at source bit #lsb to destination starting at bit 0; clears all other rd bits | |
| | Signed bitfield extract | SBFX | SBFZ rd, rn, #lsb, #width | Moves a bitfield of #width bits starting at source bit #lsb to destination starting at bit 0; sign extends the result | |
| Bit/Byte ops | Count leading sign | CLS | CLS rd, rm | Count leading sign bits | |
| | Count leading sign | CLZ | CLZ rd, rm | Count leading zero bits | |
| | Reverse bit | RBIT | RBIT rd, rm | Reverse bit order | |
| | Reverse byte | REV | REV rd, rm | Reverse byte order | |
| | Reverse byte in half word | REV16 | REV16 rd, rm | Reverse byte order on each half word | |
| | Reverse byte in word | REV32 | REV32 xd, xm | Reverse byte order on each word | |
| Load and Store operations | Store single register | STR | rt, [addr] | Mem[addr] = rt | |
| | Subtype byte | STRB | wt, [addr] | Byte[addr] = wt<7:0> | |
| | Subtype half word | STRH | wt, [addr] | HalfWord[addr] = wt<15:0> | |
| | Store register pair | STP | STP rt, rm, [addr] | Stores rt and rm in consecutive positions starting at addr | |
| | Load single register | LDR | LDR rt, [addr] | rt = Mem[addr] | |
| | Sub-type byte | LDRB | LDRB wt, [addr] | wt = Byte[addr] (only 32-byte containers) | |
| | Sub-type signed byte | LDRSB | LDRSB rt, [addr] | rt = Sbyte[addr] (signed byte) | |
| | Sub-type half word | LDRH | LDRH wt, [addr] | wt = HalfWord[addr] (only 32-byte containers) | |
| | Sub-type signed half word | LDRSH | LDRSH rt, [addr] | rt = Mem[addr] (load one half word, signed) | |
| | Sub-type signed word | LDRSW | LDRSW xt, [addr] | xt = Sword[addr] (signed word, only for 64-byte containers) | |
| | Load register pair | LDP | LDP rt, rm, [addr] | Loads rt and rm from consecutive positions starting at addr | |

| | Instruction | Mnemonic | Syntax | Explanation | Flags |
|------------------------|--|----------|------------------------|--|-------|
| Branch ops | Branch | B | B target | Jump to target | |
| | Conditional branch | B.CC | B.cc target | If (cc) jump to target | |
| | Compare and branch if zero | CBZ | CBZ rd, target | If (rd=0) jump to target | |
| | Compare and branch if not zero | CBNZ | CBNZ rd, target | If (rd≠0) jump to target | |
| Conditional operations | Conditional select | CSEL | CSEL rd, rn, rm, cc | If (cc) rd = rn else rd = rm | |
| | with increment, | CSINC | CSINC rd, rn, rm, cc | If (cc) rd = rn else rd = rn+1 | |
| | with negate, | CSNEG | CSNEG rd, rn, rm, cc | If (cc) rd = rn else rd = -rm | |
| | with invert | CSINV | CSINV rd, rn, rm, cc | If (cc) rd = rn else rd = ~rm | |
| | Conditional set | CSET | CSET rd, cc | If (cc) rd = 1 else rd = 0 | |
| | with mask, | CSETM | CSETM rd, cc | If (cc) rd = -1 else rd = 0 | |
| | with increment, | CINC | CINC rd, rn, cc | If (cc) rd = rn+1 else rd = rn | |
| | with negate, | CNEG | CNEG rd, rn, cc | If (cc) rd = -rn else rd = rn | |
| | with invert | CINV | CINV rd, rn, cc | If (cc) rd = ~rn else rd = rn | |
| Compare ops | Compare | CMP | CMP rd, op2 | Rd - op2 | Yes |
| | with negative | CMN | CMN rd, op2 | rd - (-op2) | Yes |
| | Conditional compare | CCMP | CCMP rd, rn, #imm4, cc | If (cc) NZCV = CMP(rd,rn) else NZCV = #imm4 | Yes |
| | with negative | CCMN | CCMP rd, rn, #imm4, cc | If (cc) NZCV = CMP(rd,-rn) else NZCV = #imm4 | Yes |
| | Note: for these instructions rn can also be an #im5 (5-bit unsigned immediate value 0..32) | | | | |

Aarch64 accessory information

| Condition codes (magnitude of operands) | | | Condition codes (direct flags) | | |
|---|-------------------------------|------------------|--------------------------------|------------------|-------|
| LO | Lower, unsigned | C = 0 | EQ | Equal | Z = 1 |
| HI | Higher, unsigned | C = 1 and Z = 0 | NE | Not equal | Z = 0 |
| LS | Lower or same, unsigned | C = 0 or Z = 1 | MI | Negative | N = 1 |
| HS | Higher or same, unsigned | C = 1 | PL | Positive or zero | N = 0 |
| LT | Less than, signed | N != V | VS | Overflow | V = 1 |
| GT | Greater than, signed | Z = 0 and N = V | VC | No overflow | V = 0 |
| LE | Less than or equal, signed | Z = 1 and N != V | CS | Carry | C = 0 |
| GE | Greater than or equal, signed | N = V | CC | No carry | C = 1 |

| Sub types (suffix of some instructions) | | | Flags set to 1 when: | |
|---|----------------------------|---------|----------------------|---|
| B/SB | byte/signed byte | 8 bits | N | the result of the last operation was negative, cleared to 0 otherwise |
| H/SH | half word/signed half word | 16 bits | Z | the result of the last operation was zero, cleared to 0 otherwise |
| W/SW | word/signed word | 32 bits | C | the last operation resulted in a carry, cleared to 0 otherwise |
| | | | V | the last operation caused overflow, cleared to 0 otherwise |

| Sizes, in Assembly and C | | | Addressing modes (base: register; offset: register or immediate) | |
|--------------------------|-------------|-----------|--|--|
| 8 | byte | char | [base] | MEM[base] |
| 16 | Half word | short int | [base, offset] | MEM[base+offset] |
| 32 | word | int | [base, offset]! | MEM[base+offset] then base = base + offset (pre indexed) |
| 64 | double word | long int | [base], offset | MEM[base] then base = base + offset (post indexed) |
| 128 | quad word | - | | |

| Calling convention (register use) | | Op2 processing (applied to Op2 before anything else) | |
|-----------------------------------|---------------------------------|--|---|
| Params: | X0..X7; Result: X0 | LSL LSR ASR | #imm |
| Reserved: | X8, X16..X18 (do not use these) | SXTW / SXTB | {#imm2} Sign extension/Sign extension after LSL #imm2 |
| Unprotected: | X9..X15 (callee may corrupt) | | |
| Protected: | X19..X28 (callee must preserve) | | |

Aarch64 floating point instructions

General concepts and conventions

Registers: Di (double precision: 64-bit, c:double), Si (single precision: 32-bit, c:float), Hi (half precision: 16-bit, c:non standard), i:0..31

Call convention: R0..R7 – arguments, R0 – result; R={D,S,H}; R8:R15 should be preserved by callee

Containers: r = {D,S,H}; #immn = n-bit constant

| | Instruction | Mnemonic | Syntax | Explanation | Flags |
|--|----------------------|----------|-----------------------|-------------------------------|-------|
| Arithmetic and math operations | Addition | FADD | FADD rd, rn, rm | $rd = rn + rm$ | YES |
| | Subtraction | FSUB | FSUB rd, rn, rm | $rd = rn - rm$ | YES |
| | Multiply | FMUL | FMUL rd, rn, rm | $rd = rn \times rm$ | YES |
| | Multiply and neg | FNMUL | FNMUL rd, rn, rm | $Rd = -(rn \times rm)$ | YES |
| | Multiply and add | FMADD | FMADD rd, rn, rm, ra | $rd = ra + (rn \times rm)$ | YES |
| | Multiply and add neg | FNMADD | FNMADD rd, rn, rm, ra | $Rd = -(ra + (rn \times rm))$ | YES |
| | Multiply and sub | FMSUB | FMSUB rd, rn, rm, ra | $rd = ra - (rn \times rm)$ | YES |
| | Multiply and sub neg | FNMSUB | FNMSUB rd, rn, rm, ra | $rd = (rn \times rm) - ra$ | YES |
| | Divide | FDIV | FDIV rd, rn, rm | $rd = rn / rm$ | YES |
| | Negation | FNEG | FNEG rd, rn | $rd = -rn$ | YES |
| | Absolute value | FABS | FABS rd, rn | $rd = rn $ | YES |
| | Maximum | FMAX | FMAX rd, rn, rm | $rd = \max(rn, rm)$ | YES |
| | Minimum | FMIN | FMIN rd, rn, rm | $rd = \min(rn, rm)$ | YES |
| | Square root | FSQRT | FSQRT rd, rn | $rd = \sqrt{rn}$ | YES |
| | Round to integer | FRINTI | FRINTI rd, rn | $Rd = \text{round}(rn)$ | YES |
| Note: r={D,S,H} but operands and result must be of same type | | | | | |

| | | | | | |
|---------------|---|-----------|----------------------|--|--|
| Data movement | Between registers of equal size | FMOV | FMOV rd, rn | $rd = rn$ (rd={D,S,H,X,W}; rn={D,S,H,X,W,WZR,XZR}) | |
| | immediate FP constant | FMOV | FMOV rd, #immf | $rd = \text{immf}$ (can be in scientific notation, ex: 1.2; 5.0e-3) | |
| | Between registers and memory | LDR/STR | LDR/STR rt, [addr] | $rt = \text{Mem}[addr]$; $\text{Mem}[addr] = rt$ (scaled address) | |
| | unscaled address offset | LDUR/STUR | LDUR/STR rt, [addr] | $rt = \text{Mem}[addr]$; $\text{Mem}[addr] = rt$ (unscaled address) | |
| | Load/store pair of registers | LDP/STP | LDP rt, rm, [addr] | Load/store rt and rm from/to consecutive addresses | |
| Conditional | Conditional | FCSEL | FCSEL rd, rn, rm, cc | If (cc) $rd = rn$ else $rd = rm$ | |
| | Notes: data movement with decreasing precision may lead to rounding or NaN; FP constants #immf do not include 0.0 | | | | |

| | | | | | |
|--|---------------------|-------|-------------------------|--|-----|
| Comparisons | Compare | FCMP | FCMP rn, rm | $\text{NZCV} = \text{compare}(rn, rm)$ | Yes |
| | with zero | FCMP | FCMP rd, #0.0 | $\text{NZCV} = \text{compare}(rn, 0)$ | Yes |
| | Conditional compare | FCCMP | FCCMP rn, rm, #imm4, cc | If (cc) $\text{NZCV} = \text{compare}(rn, rm)$ else $\text{NZCV} = \#imm4$ | Yes |
| Note: comparison of FP numbers can lead to wrong conclusions on very similar operands due to rounding errors | | | | | |

| | | | | | |
|---|------------------------|--------|---------------|----------------------------------|--|
| Format conversion | Between FP registers | FCVT | FCVT rd, rn | $rd = rn$ (r={D,S,H}) | |
| | signed integer to FP | SCVTF | SCVTF rd, rn | $rd = rn$ (rd={D,S,H}, rn={X,W}) | |
| | unsigned integer to FP | UCVTF | UCVTF rd, rn | $rd = rn$ (rd={D,S,H}, rn={X,W}) | |
| | FP to signed integer | FCVTNS | FCVTNS rd, rn | $rd = rn$ (rd={X,W}, rn={D,S,H}) | |
| | FP to unsigned integer | FCVTNU | FCVTNU rd, rn | $rd = rn$ (rd={X,W}, rn={D,S,H}) | |
| Note: conversion to integer can lead to an exception if the destination container does not have the required size | | | | | |

Aarch64 Advanced SIMD instructions (NEON) – work in progress

General concepts and conventions

Registers: Vi (128-bit – quadword), i:0..31; each register can be divided in lanes of {8,16,32,64} bits ({B,H,S,D})

Syntax for register structure: Vi.nk, i=register number, k=lane type {B,H,S,D}, n=number of lanes; nk={8B,16B,4H,8H,2S,4S,1D,2D}

Syntax for register element: Vi.k[n], i=register number, k=lane type {B,H,S,D}, n=element number

Examples: V3.4S = V3 structured in 4 lanes of 32 bits; V5.B[0] = rightmost byte of V5 (least significant byte)

Register space shared with FP registers (e.g. V0.D[0]=D0); Qi represents the unstructured 128-bit register space Vi

Instructions: not necessarily new mnemonics but new syntax and behaviour; if necessary, prefix {U,S,F} indicates operands data type

Examples: ADD W0,W1,W2 (signed integer addition); ADD V0.4S,V1.4S,V2.4S (signed 4-component integer vector addition)

Can operate vectors, scalars and in some cases vectors with scalars

The following tables contain only new instructions or examples. Most of the old instructions are still valid but adopt a vector behaviour

| | Instruction | Mnemonic | Syntax | Explanation | Flags |
|---------------|--|----------|----------------------|--|-------|
| Data movement | Duplicate vector element | DUP | DUP Vd.nk, Vs.k[m] | Replicate single Vs element to all elements of Vd | |
| | scalar element | DUP | DUP Vd.nk, Scl | Replicate scalar Scl to all elements of Vd (S=lsbits of {X,W}) | |
| | Insert vector element | INS | INS Vd.k[i], Vs.r[j] | Copy element r[j] of Vs to element k[i] of Vd | |
| | scalar element | INS | INS Vd.k[i], Scl | Copy scalar Scl to element k[i] of Vd (S=lsbits of {X,W}) | |
| | Note: 64-bit scalar can only be used with 64-bit lanes, 32-bit scalar can be used with 32/16/8-bit lanes | | | | |

| | | | | | |
|--|----------------------------------|--------|----------------------------|---|--|
| Vector arithmetic | Signed long (add as example) | SADDL | SADDL Vd.nk, Vs.nj, Vr.np | k has double size of j/p (ex: SADDL V0.2D,V1.2S,V2.2S) | |
| | for higher lanes | SADDL2 | SADDL2 Vd.nk, Vs.nj, Vr.np | The same, but taking the most significant lanes of operands | |
| | for wide operands | SADDW | SADDW Vd.nk, Vs.nj, Vr.np | k and j have the double size of p | |
| | wide operands, higher lanes | SADDW2 | SADDW2 Vd.nk, Vs.nj, Vr.np | The same, but taking the most significant lanes of op2 | |
| | Narrow operands (sub as example) | SUBHN | | | |
| Note: variants have different suffixes – {L,W,N,SQ,UQ,P} (long, wide, narrow, unsigned/signed saturating, pairing) | | | | | |

| | | | | | |
|-----------|--|--------|-------------------|---|--|
| Reduction | Add across lanes | ADDV | ADDV Scl, Vs.nk | Add all elements of Vs (ex: ADDV W0, V2.4S) | |
| | Signed long add across lanes | SADDLV | SADDLV Scl, Vs.nk | The same but Scl larger than k (ex: SADDLV X0, V2.4S) | |
| | Signed maximum across lanes | SMAV | SMAV Scl, Vs.nk | Maximum goes to scalar Scl | |
| | minimum | SMINV | SMINV Scl, Vs.nk | Minimum goes to scalar Scl | |
| | Note: prefix {U,S,F} defines data type (ex: FMINV finds the minimum element of an FP vector) | | | | |