AArch64 most common instructions

General conventions

Containers: x (64-bit register), w (32-bit register) — if Y is present flags will be affected

rd, rn, rm: w or x registers; op2: register, modified register or #immn (n-bit immediate)

n (of n-bit immediate values) depends heavily on context. Out of bond values will generate a syntax error

	Instruction	Mnemonic	Syntax	Explanation	Flags
Arithmetic operations	Addition	ADD{S}	ADD{S} rd, rn, op2	rd = rn + op2	Y
	Subtraction	SUB{S}	SUB{S} rd, rn, op2	rd = rn – op2	Y
	Negation	NEG{S}	NEG{S} rd, op2	rd = –op2	Y
	with carry	NGC{S}	NGC{S} rd, rm	rd = –rm – ~C	Y
	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, wn, wm	xd = wm x wn (signed operands)	
	Signed multiply high	SMULH	SMULH xd, xn, xm	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	Y
	Bitwise AND with neg	BIC	BIC{S} rd, rn, op2	rd = rn & ~op2	Y
	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, op2	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 (considering the register as a ring)	
	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	Y
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	UBFX 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	SBFX 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>	
	unscaled address offset	STUR	STUR rt, [addr]	Mem[addr] = rt (unscaled address)	
	Store register pair	STP	STP rt, rm, [addr]	Stores rt and rm in consecutive addresses 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-bit 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-bit 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-bit containers)	
	unscaled address offset	LDUR	LDUR rt, [addr]	rt = Mem[addr] (unscaled address)	
	Load register pair	LDP	LDP rt, rm, [addr]	Loads rt and rm from consecutive addresses starting at addr	

	Instruction	Mnemonic	Syntax	Explanation	Flags
Branch operations	Branch	B	B target	Jump to target	
	Branch and link	BL	BL target	Writes the addr of the next instr to X30 and jumps to target	
	Return	RET	RET {Xm}	Returns from sub-routine jumping through register Xm (default: X30)	
	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 = rm+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 #imm5 (5-bit unsigned immediate value 0..31)				

AArch64 accessory information

Condition codes (magnitude of operands)		
LO	Lower, unsigned	C = 0
HI	Higher, unsigned	C = 1 and Z = 0
LS	Lower or same, unsigned	C = 0 or Z = 1
HS	Higher or same, unsigned	C = 1
LT	Less than, signed	N != V
GT	Greater than, signed	Z = 0 and N = V
LE	Less than or equal, signed	Z = 1 and N != V
GE	Greater than or equal, signed	N = V

Condition codes (direct flags)		
EQ	Equal	Z = 1
NE	Not equal	Z = 0
MI	Negative	N = 1
PL	Positive or zero	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
CS	Carry	C = 0
CC	No carry	C = 1

Sub types (suffix of some instructions)		
B/SB	byte/signed byte	8 bits
H/SH	half word/signed half word	16 bits
W/SW	word/signed word	32 bits

Flags set to 1 when:	
N	the result of the last operation was negative, cleared to 0 otherwise
Z	the result of the last operation was zero, cleared to 0 otherwise
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		
8	byte	char
16	Half word	short int
32	word	int
64	double word	long int
128	quad word	-

Addressing modes (base: register; offset: register or immediate)		
[base]	MEM[base]	
[base, offset]	MEM[base+offset]	
[base, offset]!	MEM[base+offset] then base = base + offset	(pre indexed)
[base], offset	MEM[base] then base = base + offset	(post indexed)

Calling convention (register use)
Params: X0..X7; Result: X0
Reserved: X8, X16..X18 (do not use these)
Unprotected: X9..X15 (callee may corrupt)
Protected: X19..X28 (callee must preserve)

Op2 processing (applied to Op2 before anything else)	
LSL|LSR|ASR #imm6	
SXTW / SXTB {#imm2}	Sign extension/Sign extension after LSL #imm2

AArch64 floating point instructions

General concepts and conventions
Registers: Di (double precision: 64-bit, c:double), Si (single precision: 32-bit, c:float); i:0..31
Hi (half precision: 16-bit, c:non standard); i:0..31
Call convention: Reg0..Reg7 – arguments, Reg0 – result; Reg={D,S,H}; Reg8..Reg15 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	Y
	Subtraction	FSUB	FSUB rd, rn, rm	rd = rn − rm	Y
	Multiply	FMUL	FMUL rd, rn, rm	rd = rn x rm	Y
	Multiply and neg	FNMUL	FNMUL rd, rn, rm	rd = − (rn x rm)	Y
	Multiply and add	FMADD	FMADD rd, rn, rm, ra	rd = ra + (rn x rm)	Y
	Multiply and add neg	FNMADD	FNMADD rd, rn, rm, ra	rd = − (ra + (rn x rm))	Y
	Multiply and sub	FMSUB	FMSUB rd, rn, rm, ra	rd = ra − (rn x rm)	Y
	Multiply and sub neg	FNMSUB	FNMSUB rd, rn, rm, ra	rd = (rn x rm) − ra	Y
	Divide	FDIV	FDIV rd, rn, rm	rd = rn / rm	Y
	Negation	FNEG	FNEG rd, rn	rd = − rn	Y
	Absolute value	FABS	FABS rd, rn	rd = |rn|	Y
	Maximum	FMAX	FMAX rd, rn, rm	rd = max(rn,rm)	Y
	Minimum	FMIN	FMIN rd, rn, rm	rd = min(rn,rm)	Y
	Square root	FSQRT	FSQRT rd, rn	rd = sqrt(rn)	Y
	Round to integer	FRINTI	FRINTI rd, rn	rd = round(rn)	Y
	Note: r={D,S,H} but operands and result must be of same type				
Data moves	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})	
	Conditional select	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 Data movement to/from memory is still valid (e.g. LDR/STR, etc.)				
Comparisons	Compare	FCMP	FCMP rn, rm	NZCV = compare(rn,rm)	Y
	with zero	FCMP	FCMP rn, #0.0	NZCV = compare(rn,0)	Y
	Conditional compare	FCCMP	FCCMP rn, rm, #imm4, cc	If (cc) NZCV = compare(rn,rm) else NZCV = #imm4	Y
	Notes: comparison of FP numbers can lead to wrong conclusions on very similar operands due to rounding errors In general flag behaviour is similar to the integer compares. See table below.				
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				

Flag behaviour on integer and floating point compares		
cc	CMP Meaning	FCMP meaning
EQ	Equal	Equal
NE	Not equal	Not equal (or unordered)
HI	Unsigned greater than	Greater than (or unordered)
HS	Unsigned greater than or equal to	Greater than or equal to (or unordered)
LO	Unsigned less than	Less than
LS	Unsigned less than or equal to	Less than or equal to
GT	Signed greater than	Greater than
GE	Signed greater than or equal to	Greater than or equal to
LT	Signed less than	Less than (or unordered)
LE	Signed less than or equal to	Less than or equal to (or unordered)
VS	Signed overflow	Unordered (at least one argument was NaN)

AArch64 Advanced SIMD instructions (NEON)

General concepts and conventions

Vector Registers: Vi (128-bit – quadword), i:0..31; each reg can be structured in lanes of {8,16,32,64} bits {B,H,S,D}
Syntax for structure: Vi.nk, i=reg number, n=nbr of lanes, k=lane type {B,H,S,D}; 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)

Scalar Registers (Scl): Qi(128-bit), Di(64-bit), Si(32-bit), Hi(16-bit), Bi(8-bit); Shared with FP registers

Instructions: not necessarily new mnemonics but new syntax and behaviour
Can operate vectors, scalars and in some cases scalars with vectors
Examples: ADD W0,W1,W2 (signed integer addition); ADD V0.4S,V1.4S,V2.4S (signed 4-component integer vector addition)

The following tables contain only new instructions. Most of the classic instructions, for both integer and FP data, are still valid but adopt the new syntax and behaviour.
Variants can have different suffixes – {L,W,N,P} (long, wide, narrow, pairing)
Prefix F for floating point data types; prefixes {SQ,UQ} for integer signed/unsigned saturating arithmetic

	Instruction	Mnemonic	Syntax	Explanation
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})
	Extract narrow	XTN	XTN Vd.nk, Vs.mj	dim(j) = 2 x dim(k)
	for higher lanes	XTN2	XTN2 Vd.nk, Vs.mj	The same, but using the most significant lanes of Vd
	Note: 64-bit scalar can only be used with 64-bit lanes, 32-bit scalar can be used with 32/16/8-bit lanes			
	Signed move to scalar register	SMOV	SMOV Rd, Vn.T[i]	Copy vector element to register, sign extended (dim R >= dim T)
	Unsigned	UMOV	UMOV Rd, Vn.T[i]	Copy vector element to register, unsigned (dim R >= dim T)
Vector arithmetic and logic	Signed long (add as example)	SADDL	SADDL Vd.nk, Vs.nj, Vr.np	dim(k) = 2 x dim(j,p) (ex: SADDL V0.2D,V1.2S,V2.2S)
	for higher lanes	SADDL2	SADDL2 Vd.nk, Vs.nj, Vr.np	The same, but using the most significant lanes of Vs and Vr
	for wide operands	SADDW	SADDW Vd.nk, Vs.nj, Vr.np	dim(k,j) = 2 x dim(p)
	wide operands, higher lanes	SADDW2	SADDW2 Vd.nk, Vs.nj, Vr.np	The same, but using the most significant lanes of Vr
	Narrow operands (sub as example)	SUBHN	SUBHN Vd.nk, Vs.nj, Vr.np	dim(j,p) = 2 x dim(k)
	Paired (ADD as example)	ADDP	ADDP Vd.nk, Vs.nj, Vr.np	Operate adjacent register pairs
	Paired FP add	FADDP	FADDP Vd.nk, Vs.nj, Vr.np	Operate adjacent register pairs
	Shift element left	SHL	SHL Vd.nk, Vs.nj, #imm	Shift left each vector element #imm bits
	Signed shift right	SSHR	SSHR Vd.nk, Vs.nj, #imm	Shift right each vector element, sign extended, #imm bits
	unsigned	USHR	USHR Vd.nk, Vs.nj, #imm	The same but unsigned
	Bit select	BSL	BSL Vd.nk, Vs.nj, Vr.np	Select bits from Vs or Vr depending on bits of Vd (1:Vs, 0:Vr)
	Reverse elements	REV64	REV64 Vd.nk, Vs.nj	Reverse elements in 64-bit doublewords
	Other arithmetic instructions: ABS, MUL, NEG, SMAX, SUB, UMIN, FADD, etc. adopt a vector syntax and behaviour Other logic instructions: AND, BIC, EOR, NOT, ORN, ORR, REV32, REV16, etc. do the same Format conversions from/to floating point adopt a vector behaviour: UCVTF, SCVTF, FCVTNU, FCVTNS (ex: UCVTF V2.4S, V1.4S)			
Reduction	Add across lanes	ADDV	ADDV Scl, Vs.nk	Add all elements of Vs into a scalar (ex: ADDV S0, V2.4S)
	Signed long add across lanes	SADDLV	SADDLV Scl, Vs.nk	The same but dim(Scl) larger than k (ex: SADDLV D0, V2.4S)
	Paired FP add	FADDP	FADDP Scl, Vs.nk	Add a pair of elements of Vs into a scalar (ex: FADDP D2, V1.2D)
	Signed maximum across lanes	SMAXV	SMAXV Scl, Vs.nk	Maximum goes to scalar Scl
	minimum	SMINV	SMINV Scl, Vs.nk	Minimum goes to scalar Scl
	Notes: prefix {U,S,F} defines data type (ex: FMINV finds the minimum element of an FP vector) FP add across lanes is illegal			
Compare	Compare bitwise vector	CMcc	CMcc Vd.nk, Vn.nj, Vm.np	if true Vd.k[i]=-1 (all ones) else Vd.k[i]=0
	with zero	CMcc	CMcc Vd.nk, Vn.nj, #0	Compare vector with zero cc=default conditions+{LE,LT}
	FP compare vector	FCMcc	FCMcc Vd.nk, Vn.nj, Vm.np	The same, but for vectors of FP elements
	Notes: default conditions cc={EQ,GE,GT,HS,HI} conditions {LS,LE,LO,LT} are achieved by reversing the operands and using the opposite condition Flags NZCV are not afected			