## Aarch64 V8 most common instructions (Work in Progress - V5.0)

General conventions

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

Containers: x (64-bit register), w (32-bit register)

Marketion		To de contrat on		S t	Full could be	<b>51</b>
Settraction		Instruction	Mnemonic	Syntax	Explanation	Flags
New						+
					·	
Unitargued multituly   Mail					· · · · · · · · · · · · · · · · · · ·	+
						{Yes}
Signed multiply brigh   SMULL   SMULL Hod, xx, xx, xx   xx   xx   xx   xx   xx						
Suppos multiply high   Simple multiply migh		- '				
	Cons					
	-ati				xd = <127:64> of xn x xm (signed operands)	
	ber		MADD	MADD rd, rn, rm, ra		
	٢			MSUB rd, rn, rm, ra	rd = ra - (rn x rm)	
	Je t	Multiply and neg	MNEG		Rd = -(rn x rm)	
United Multiply and add long	ij		UMADDL	UMADDL xd, wn, wm, xa	xd = xa + (wm x wn)	
Signed multiply and sub long	Αr	Unsigned multiply and sub long	UMSUBL	UMSUBL xd, wn, wm, xa	xd = xa - (wm x wn)	
Signed multiply and neglong		Unsigned multiply and neg long	UMNEGL	UMNEGL xd, wn, wn	$Xd = -(wm \times wn)$	
Signed Multiply and neg long   SMMCCL   SMMCCL xd, xm, xm   Xd = - (um x vm)		Signed multiply and add long	SMADDL	SMADDL xd, wn, wm, xa	xd = xa + (wm x wn)	
Standard divide		Signed multiply and sub long	SMSUBL	SMSUBL xd, wn, wm, xa	xd = xa - (wm x wn)	
Signed divide   Solly   Solly rd, rn, rm   rd = rn / rm		Signed multiply and neg long	SMNEGL	SMNEGL xd, wn, wm	$Xd = - (wm \times wn)$	
Note: the remainder may be computed using the MSUB instruction as numerator - (quotient x denominator)    Description   AND		Unsigned divide	UDIV	UDIV rd, rn, rm	rd = rn / rm	
Bitwise AND   AND   AND   AND   AND   String or   First or   First or   String or   Stri		Signed divide	SDIV	SDIV rd, rn, rm	rd = rn / rm	
Situise AND with neg   BIC   BIC(S) rd, rn, op2   rd = rn & -op2   (Yes)		Note: the remainder may be comput	ted using t	he MSUB instruction as numerat	or – (quotient x denominator)	
Situise AND with neg   BIC   BIC(S) rd, rn, op2   rd = rn & -op2   (Yes)		hRitwise AND	ΔND	AND(S) rd rn on2	rd = rn & on?	[Ves]
Bitwise OR With neg ORN ORR rd, rn, op2 rd = rn   op2  Bitwise OR with neg ORN ORR rd, rn, op2 rd = rn   op2  Bitwise OR with neg ORN ORR rd, rn, op2 rd = rn   op2  Bitwise OR With neg EON EON 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  Bitwise XOR with neg EON EON rd, rn, op2 rd = rn = op2  Bitwise XOR with neg EON EON rd, rn, op2 rd = rn = op2  Bitwise XOR with neg EON EON rd, rn, op2 rd = rn = op2  Bitwise XOR with neg EON EON rd, rn, op2 rd = rn = op2  Bitwise XOR with neg EON EON rd, rn, op2 logical shift right (stuffing zeros enter from right)  Logical shift right LSR LSR rd, rn, rn Logical shift right (stuffing zeros enter from left)  Arithmetic shift right ASR ASR rd, rn, op2 Arithmetic shift right (preserves sign)  Rotate right Rore right ROR ROR ROR ROR ROR rn, op2 Rotate right (carry not involved)  Rove to register register register register rn, op2 rd = op2  Test bits TST TST TST rn, op2 rd = op2  Test bits TST TST TST rn, op2 rd = op2  Test bits TST TST TST RN, #Noth Row rn, #Isb, #Width Roves a bitfield of #Width bits starting at source bit 8 to destination starting at bit #Isb  Bitfield extract UBFX UBFZ rd, rn, #Isb, #Width Roves a bitfield of #Width bits starting at source bit 8 to destination starting at bit 8; clears all other rd bits Signed bitfield extract SIGN Row rn, #Isb, #Width Roves a bitfield of #Width bits starting at source bit #Isb to destination starting at bit 8; clears all other rd bits Signed bitfield extract SIGN Row rn, #Isb, #Width Roves a bitfield of #Width bits starting at source bit #Isb to destination starting at bit 8; clears all other rd bits Signed bitfield extract SIGN Row rn, #Isb, #Width Roves a bitfield of #Width bits starting at source bit #Isb to destination starting at bit 8; clears all other rd bits Signed bit 8; clears all other rd bits Row rn, #Isb, #Width Rover and						
Bitwise OR with neg						fies
Bitwise XOR  Bitwise XOR with neg  EOR  EOR rd, rn, op2  rd = rn @	Lons					
Bitwise XOR with neg	lati.					
Count leading sign   CLS   CLS rd, rn   Count leading sign   CLS   CLS rd, rn   Count leading sign   CLS   CLS rd, rn   Count leading sign   CLZ   CLZ rd, rn   Count leading sign   CLZ   CLZ rd, rn   Count leading sign   CLZ   CLZ rd, rn   Reverse byte in half word   REV16   REV16 rd, rn   Reverse byte order   Reverse byte in word   REV16   REV16 rd, rn   Reverse byte order   Reverse byte in word   STRH   Mt, [addr]   Mt = Mt[Addr]   Stores rt and rm in consecutive positions starting at addr   CLS   CLS Rd rd]   Rev = Store signed word   CLDRSH   LDRSH rt, [addr]   rt = Sword[addr] (signed word, only for 64-byte containers)   CDRSH rt, [addr]   rt = Sword[addr] (signed word, only for 64-byte containers)   CLS Store register   CLDRSH rt, [addr]   rt = Sword[addr] (signed word, only for 64-byte containers)   CLDRSH rt, [addr]   rt = Sword[addr] (signed word, only for 64-byte containers)   CLDRSH rt   CLDR	be				1	
Example   Logical shift right   LSR   LSR rd, rn, rm   Logical shift right (stuffing zeros enter from left)	ال				·	
Move to register MOV MOV rd, op2 rd = op2  Move to register, neg MVN MVN rd, op2 rd = -op2  Test bits TST TIST rn, op2 rd = op2  Test bits TST TIST rn, op2 rn & op2  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  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; clears all other rd bits  Count leading sign CLS CLS rd, rm Count leading sign bits  Count leading sign CLZ CLZ rd, rm Count leading sign bits  Count leading sign CLZ CLZ rd, rm Reverse bit order  Reverse byte  Reverse byte in #REVI REVI rd, rm Reverse byte order  Reverse byte in half word REV16 REV16 rd, rm Reverse byte order  Reverse byte in word REV32 REV32 xd, xm Reverse byte order on each half word  Reverse byte in word REV32 REV32 xd, xm Reverse byte order on each word  Store single register STR rt, [addr] Mem[addr] = rt  Subtype byte STRB wt, [addr] Byte[addr] = wt-15:0>  Subtype byte STRB wt, [addr] the #New16addr] rt = Mem[addr]  Sub-type signed byte LDRB LDRS wt, [addr] rt = Sbyte[addr] (only 32-byte containers)  Sub-type signed byte CDRS LDRS rt, [addr] rt = Sbyte[addr] (only 32-byte containers)  Sub-type signed half word LDRSH LDRSH rt, [addr] rt = Sword[addr] (signed word, only for 64-byte containers)  Sub-type signed word LDRSH LDRSH rt, [addr] rt = Sword[addr] (signed word, only for 64-byte containers)	gic					
Move to register MOV MOV rd, op2 rd = op2  Move to register, neg MVN MVN rd, op2 rd = -op2  Test bits TST TIST rn, op2 rd = op2  Test bits TST TIST rn, op2 rn & op2  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  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; clears all other rd bits  Count leading sign CLS CLS rd, rm Count leading sign bits  Count leading sign CLZ CLZ rd, rm Count leading sign bits  Count leading sign CLZ CLZ rd, rm Reverse bit order  Reverse byte  Reverse byte in #REVI REVI rd, rm Reverse byte order  Reverse byte in half word REV16 REV16 rd, rm Reverse byte order  Reverse byte in word REV32 REV32 xd, xm Reverse byte order on each half word  Reverse byte in word REV32 REV32 xd, xm Reverse byte order on each word  Store single register STR rt, [addr] Mem[addr] = rt  Subtype byte STRB wt, [addr] Byte[addr] = wt-15:0>  Subtype byte STRB wt, [addr] the #New16addr] rt = Mem[addr]  Sub-type signed byte LDRB LDRS wt, [addr] rt = Sbyte[addr] (only 32-byte containers)  Sub-type signed byte CDRS LDRS rt, [addr] rt = Sbyte[addr] (only 32-byte containers)  Sub-type signed half word LDRSH LDRSH rt, [addr] rt = Sword[addr] (signed word, only for 64-byte containers)  Sub-type signed word LDRSH LDRSH rt, [addr] rt = Sword[addr] (signed word, only for 64-byte containers)	اي	Logical shift right				
Move to register MOV MOV rd, op2 rd = op2  Move to register, neg MVN MVN rd, op2 rd = -op2  Test bits TST TIST rn, op2 rd = op2  Test bits TST TIST rn, op2 rn & op2  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  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; clears all other rd bits  Count leading sign CLS CLS rd, rm Count leading sign bits  Count leading sign CLZ CLZ rd, rm Count leading sign bits  Count leading sign CLZ CLZ rd, rm Reverse bit order  Reverse byte  Reverse byte in #REVI REVI rd, rm Reverse byte order  Reverse byte in half word REV16 REV16 rd, rm Reverse byte order  Reverse byte in word REV32 REV32 xd, xm Reverse byte order on each half word  Reverse byte in word REV32 REV32 xd, xm Reverse byte order on each word  Store single register STR rt, [addr] Mem[addr] = rt  Subtype byte STRB wt, [addr] Byte[addr] = wt-15:0>  Subtype byte STRB wt, [addr] the #New16addr] rt = Mem[addr]  Sub-type signed byte LDRB LDRS wt, [addr] rt = Sbyte[addr] (only 32-byte containers)  Sub-type signed byte CDRS LDRS rt, [addr] rt = Sbyte[addr] (only 32-byte containers)  Sub-type signed half word LDRSH LDRSH rt, [addr] rt = Sword[addr] (signed word, only for 64-byte containers)  Sub-type signed word LDRSH LDRSH rt, [addr] rt = Sword[addr] (signed word, only for 64-byte containers)	ise	Arithmetic shift right				
Move to register, neg Test bits TST TST rn, op2 Test bits TST TST rn, op2 Tn & op2 Test bits TST TST rn, op2 Tn & op2 Tn & op2 Test bits TST TST rn, op2 Tn & op2 The first bit dithet bits starting at buftel dot	it	Rotate right				
Test bits  TST TST n, op2  Test op2  Test op3  Test bits  TST TST n, op2  Test op3  Test op3  Test bits  TST TST n, op2  Test op3  Test bits  TST TST n, op2  Test op3  Test bits  TST TST n, op2  Test op3  Test op3  Test op3  Test op3  Test op3  Test op4  Test op3  Test op4	ω				· ·	
Bitfield insert  BFI BFI rd, rn, #lsb, #width destination starting at bit #lsb  Bitfield extract  UBFX UBFZ rd, rn, #lsb, #width 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; clears all other rd bits  Count leading sign  CLS CLS rd, rm  Count leading sign bits  Count leading sign bits  Count leading sign bits  Reverse bit  REIT RBIT rd, rm  Reverse bit order  Reverse byte  Reverse byte in half word  Reverse byte in half word  Reverse byte in half word  REV16 REV16 rd, rm  Reverse byte order on each half word  Reverse byte in word  Reverse byte in word  Reverse byte order on each word  Store single register  STR  rt, [addr]  Mem[addr] = rt  Subtype byte  STRB wt, [addr]  Subtype balf word  STRH wt, [addr]  Stores rt and rm in consecutive positions starting at addr  to sub-type signed byte  LDRS wt, [addr]  wt = Byte[addr] (signed byte)  Sub-type signed blaf word  LDRS LDRS rt, [addr]  rt = Mem[addr] (signed word, signed)  Sub-type signed word  LDRS LDRS xt, [addr]  rt = Mem[addr] (signed word, signed)  Sub-type signed word  Sub-type signed word  LDRSW LDRSW xt, [addr]  xt = Sword[addr] (signed word, only for 64-byte containers)						
Bitfield extract  UBFX  UBFZ rd, rn, #lsb, #width  destination 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  Count leading sign  CLS  CLS rd, rm  Count leading sign bits  Count leading sign bits  Count leading sign bits  Count leading sign bits  RBIT rd, rm  Reverse bit order  Reverse byte  REV rd, rm  Reverse byte order  Reverse byte order  Reverse byte order on each half word  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  Store single register  Subtype byte  Subtype byte  Subtype half word  STRH  wt, [addr]  Men[addr] = rt  Subtype half word  Store register pair  Store register pair  Store register pair  Store single register  LDR  LDR rt, [addr]  Sub-type byte  Sub-type signed byte  LDRSB  LDRSB rt, [addr]  wt = MalfWord[addr] (only 32-byte containers)  Sub-type signed half word  Sub-type signed word  LDRSH  LDRSH rt, [addr]  wt = MalfWord[addr] (load one half word, signed)  Sub-type signed word  LDRSH  LDRSH rt, [addr]  rt = Mem[addr] (load one half word, signed)  Sub-type signed word  LDRSH  LDRSH rt, [addr]  rt = Mem[addr] (load one half word, signed)  Sub-type signed word  LDRSH  LDRSH rt, [addr]  rt = Mem[addr] (signed word, only for 64-byte containers)			TST	TST rn, op2	rn & op2	Yes
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  Count leading sign CLZ CLZ 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  Store single register  Subtype byte STRB wt, [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 single register  STP rt, rm, [addr] Stores rt and rm in consecutive positions starting at addr  Load single register  SUB LDRB wt, [addr] wt = Byte[addr] (only 32-byte containers)  Sub-type signed byte LDRS LDRS rt, [addr] rt = Sbyte[addr] (only 32-byte containers)  Sub-type signed word LDRH LDRH wt, [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)	sdo	Bitfield insert	BFI	BFI rd, rn, #lsb, #width		
Count leading sign   CLS   CLS rd, rm   Count leading sign bits  Reverse bit   RBIT   RBIT   RBIT rd, rm   Reverse byte order  Reverse byte order  Reverse byte in half word   REV16   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   Store single register   STR   rt, [addr]   Mem[addr] = rt   Subtype byte   Subtype half word   Store register pair   STR   str, [addr]   HalfWord[addr] = wt<7:0>  Store register pair   STP   STP rt, rm, [addr]   Store register pair   Store register   STP   STP rt, rm, [addr]   Store stand rm in consecutive positions starting at addr   Load single register   LDR   LDR   LDR   LDR   LDR   LDR   LDR   Sub-type byte   Sub-type signed byte   LDRS   LDRS   LDRS   LDRS   LDRS   LDRS   LDRH   LDRH   LDRH   LDRH   LDRH   LDRH   LDRH   LDRH   LDRSH   LD	ield		UBFX	UBFZ rd, rn, #lsb, #width		
Count leading sign  CLZ CLZ rd, rm  Count leading zero bits  Reverse bit  Reverse bit  Reverse byte  Reverse byte  Reverse byte  Reverse byte in half word  Reverse byte in word  Reverse byte order on each half word  Reverse byte order on each word  Store single register  STR rt, [addr]  Mem[addr] = rt  Subtype byte  Store register pair  Store register pair  Store register pair  Store register  LDR LDR rt, [addr]  Sub-type byte  Sub-type byte  Sub-type signed byte  LDRSB LDRSB rt, [addr]  Tr = Swyte[addr] (signed byte)  Sub-type signed half word  Sub-type signed word  LDRSH LDRSH rt, [addr]  Tr = 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)	Bitf	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	
Count leading sign  CLZ CLZ rd, rm  Count leading zero bits  Reverse bit  Reverse bit  Reverse byte  Reverse byte  Reverse byte  Reverse byte in half word  Reverse byte in word  Reverse byte order on each half word  Reverse byte order on each word  Store single register  STR rt, [addr]  Mem[addr] = rt  Subtype byte  Store register pair  Store register pair  Store register pair  Store register  LDR LDR rt, [addr]  Sub-type byte  Sub-type byte  Sub-type signed byte  LDRSB LDRSB rt, [addr]  Tr = Swyte[addr] (signed byte)  Sub-type signed half word  Sub-type signed word  LDRSH LDRSH rt, [addr]  Tr = 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)		Count leading sign	CLS	CLS rd. rm	Count leading sign bits	$\top$
Reverse bit RBIT RBIT rd, rm Reverse byte order Reverse byte in half word Reverse byte in word Reverse byte in word REV32 REV32 xd, xm Reverse byte order on each half word Reverse byte in word REV32 REV32 xd, xm Reverse byte order on each word  Store single register STR rt, [addr] Subtype byte Subtype half word STRH wt, [addr] Store register pair STP STP rt, rm, [addr] Store register pair LOAD SIDE STP STP rt, rm, [addr] Sub-type byte Sub-type byte SUB-type signed byte SUB-type signed word SUB-type signed word SUB-type signed word SUB-SSW LDRSH L	SG			· ·		+
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  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 LDRSH LDRSH rt, [addr] rt = Sbyte[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)	0		+			+
Reverse byte in half word Reverse byte in half word Reverse byte in word Reverse byte order on each half word Reverse byte in word Reverse byte order on each half word Reverse byte in word Reverse byte order on each half word Reverse byte in word Reverse byte order on each half word Reverse byte in word Reverse byte order on each half word Reverse byte in word Reverse byte order on each half word Reverse byte order on each word Reverse byte order on each half word Reverse byte order on each word) Reverse byte order on each word) Reverse byte order on each word; Subtype half word LDR Without and Reverse byte orde				· ·		+
Reverse byte in word  REV32 REV32 xd, xm  Reverse byte order on each word  Mem[addr] = rt  Subtype byte  Subtype byte  Subtype byte  Subtype half word  Str yt, [addr]  Subtype half word  Str yt, [addr]  Str yt, [addr]  Store register pair  Str yt, rm, [addr]  Load single register  LDR  LDR rt, [addr]  Sub-type byte  LDRB LDRB wt, [addr]  Sub-type signed byte  Sub-type signed byte  Sub-type signed half word  LDRSH  LDRSH rt, [addr]  rt = Mem[addr] (only 32-byte containers)  Sub-type signed half word  Sub-type signed word  LDRSH  LDRSH rt, [addr]  rt = Mem[addr] (load one half word, signed)  Sub-type signed word  Sub-type signed word  LDRSW  LDRSW xt, [addr]  xt = Sword[addr] (signed word, only for 64-byte containers)	it/	Reverse byte in half word				+
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)	ω					
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 rt = Mem[addr] Load single register LDR LDRB wt, [addr] rt = Mem[addr] (only 32-byte containers) Sub-type byte LDRB LDRSB t, [addr] rt = Sbyte[addr] (signed byte) Sub-type signed byte LDRB LDRSB rt, [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)						
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)						+
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)	ons					+
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)	ati					+
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)	per	- ·				+
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)						
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)	tor					+
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)						+
Sub-type signed word LDRSW LDRSW xt, [addr] xt = Sword[addr] (signed word, only for 64-byte containers)						+
	_oac					+
Load register pair LDP rt, rm, [addr] Loads rt and rm from consecutive positions starting at addr						+
		Loau register pair	LUP	בטיי רו, ריי, [מססר]	LUGUS IT AND THE FROM CONSECUTIVE POSITIONS STARTING AT ADDR	

	Instruction	Mnemonic	Syntax	Explanation	Flags
bs	Branch	В	B target	Jump to target	
0	Conditional branch	B.CC	B.cc target	If (cc) jump to target	
-anch	Compare and branch if zero	CBZ	CBZ rd, target	If (rd=0) jump to target	
뮵	Compare and branch if not zero	CBNZ	CBNZ rd, target	If (rd≠0) jump to target	
	Conditional select	CSEL	CSEL rd, rn, rm, cc	If (cc) rd = rn else rd = rm	
Su	with increment,	CSINC	CSINC rd, rn, rm, cc	If (cc) rd = rn else rd = rm+1	
ations	with negate,	CSNEG	CSNEG rd, rn, rm, cc	If (cc) rd = rn else rd = -rm	
Pe	with invert	CSINV	CSINV rd, rn, rm, cc	If (cc) rd = rn else rd = ~rm	
0	Conditional set	CSET	CSET rd, cc	If (cc) rd = 1 else rd = 0	
Conditional	with mask,	CSETM	CSETM rd, cc	If (cc) rd = -1 else rd = 0	
lg;t	with increment,	CINC	CINC rd, rn, cc	If (cc) rd = rn+1 else rd = rn	
S	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	СМР	CMP rd, op2	Rd - op2	Yes
ops	with negative	CMN	CMN rd, op2	rd - (-op2)	Yes
are	Conditional compare	ССМР	CCMP rd, rn, #im4, cc	If (cc) NZCV = CMP(rd,rn) else NZCV = #imm4	Yes
Comp	with negative	CCMN	CCMP rd, rn, #im4, cc	If (cc) NZCV = CMP(rd,-rn) else NZCV = #imm4	Yes
	Note: for these instructions rn ca	n also be	an #im5 (5-bit unsigned imm	ediate value 032)	

## Aarch64 V8 accessory information

Conditi	on codes (magnitude of operands)	
L0	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

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

Sub typ	es (suffix of some instruction	ıs)
B/SB	byte/signed byte	8 bits
H/SH	half word/signed half word	16 bits
W/SW	word/signed word	32 bits

Fla	gs 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
С	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: regist	er; 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: X0X7; Result: X0
Reserved: X8, X16X18 (do not use these)
Unprotected: X9X15 (callee may corrupt)
Protected: X19 . X28 (callee must preserve)

Op2 processing (applied t	o Op2 before anything else)	
LSL LSR ASR #imm		
SXTW / SXTB {#imm2}	Sign extension/Sign extension after LSL #imm2	

## Aarch64 V8 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}; R8:15 should be preserved by callee

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

Instruction	Mnemonic	Syntax	Explanation		Fla
Addition	FADD	FADD rd, rn, rm	rd = rn + rm		YE
Subtraction	FSUB	FSUB rd, rn, rm	rd = rn - rm		YE
Multiply	FMUL	FMUL rd, rn, rm	rd = rn x rm		YE
Multiply and neg	FNMUL	FNMUL rd, rn, rm	Rd = - (rn x rm)		YE
Multiply and add	FMADD	FMADD rd, rn, rm, ra	rd = ra + (rn x rm)		ΥI
Multiply and add neg	FNMADD	FNMADD rd, rn, rm, ra	Rd = - (ra + (rn x rm))		Υ
Multiply and sub	FMSUB	FMSUB rd, rn, rm, ra	rd = ra - (rn x rm)		Υ
Multiply and sub neg	FNMSUB	FNMSUB rd, rn, rm, ra	rd = (rn x rm) - ra		Υ
Divide	FDIV	FDIV rd, rn, rm	rd = rn / rm		Υ
Negation	FNEG	FNEG rd, rn	rd = - rn		Υ
Absolute value	FABS	FABS rd, rn	rd =  rn		Υ
Negation Absolute value Maximum Minimum	FMAX	FMAX rd, rn, rm	rd = max(rn,rm)		Υ
Minimum	FMIN	FMIN rd, rn, rm	rd = min(rn,rm)		Y
Square root	FSQRT	FSQRT rd, rn	rd = sqrt(rn)		Υ
Round to integer	FRINTI	FRINTI rd, rn	Rd = round(rn)		١
Round to threger					
Note: r={D,S,H} but operands a	nd result mus	t be of same type			
	nd result mus	t be of same type FMOV rd, rn	rd = rn	(r={D,S,H,X,W})	
Note: r={D,S,H} but operands a			rd = rn rt = Mem[addr]; Mem[addr] = rt	(r={D,S,H,X,W}) (scaled address)	
Note: r={D,S,H} but operands a	FMOV	FMOV rd, rn LDR/STR rt, [addr]			
Note: r={D,S,H} but operands a Between registers of equal size Between registers and memory	FMOV LDR/STR	FMOV rd, rn LDR/STR rt, [addr]	rt = Mem[addr]; Mem[addr] = rt	(scaled address)	
Note: r={D,S,H} but operands a Between registers of equal size Between registers and memory unscaled address offset	FMOV LDR/STR LDUR/STUR	FMOV rd, rn LDR/STR rt, [addr] LDR/STR rt, [addr]	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt	(scaled address)	
Note: r={D,S,H} but operands a  Between registers of equal size  Between registers and memory  unscaled address offset  Load/store pair of registers	FMOV  LDR/STR  LDUR/STUR  LDP/STP  FCSEL	FMOV rd, rn  LDR/STR rt, [addr]  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm	(scaled address)	
Note: r={D,S,H} but operands a  Between registers of equal size  Between registers and memory  unscaled address offset  Load/store pair of registers  Conditional	FMOV  LDR/STR  LDUR/STUR  LDP/STP  FCSEL	FMOV rd, rn  LDR/STR rt, [addr]  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm	(scaled address)	Y
Note: r={D,S,H} but operands a  Between registers of equal size  Between registers and memory  unscaled address offset  Load/store pair of registers  Conditional  Note: data movement with decr	FMOV LDR/STR LDUR/STUR LDP/STP FCSEL easing precis	FMOV rd, rn  LDR/STR rt, [addr]  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc  ion may lead to rounding or	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm NaN	(scaled address)	-
Note: r={D,S,H} but operands a  Between registers of equal size  Between registers and memory  unscaled address offset  Load/store pair of registers  Conditional  Note: data movement with decr	FMOV LDR/STR LDUR/STUR LDP/STP FCSEL easing precis	FMOV rd, rn  LDR/STR rt, [addr]  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc  tion may lead to rounding or	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm NaN  NZCV = compare(rn,rm)	(scaled address) (unscaled address) ve addresses	Y
Note: r={D,S,H} but operands a  Between registers of equal size  Between registers and memory  unscaled address offset  Load/store pair of registers  Conditional  Note: data movement with decr  Compare  with zero  Conditional compare	FMOV LDR/STR LDUR/STUR LDP/STP FCSEL easing precis FCMP FCMP FCMP	FMOV rd, rn  LDR/STR rt, [addr]  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc  tion may lead to rounding or  FCMP rn, rm  FCMN rd, #0.0  FCCMP rn, rm, #im4, cc	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm  NAN  NZCV = compare(rn,rm)  NZCV = compare(rn,0)	(scaled address) (unscaled address) ve addresses	Υ
Note: r={D,S,H} but operands a Between registers of equal size Between registers and memory unscaled address offset Load/store pair of registers Conditional Note: data movement with decr Compare with zero Conditional compare	FMOV LDR/STR LDUR/STUR LDP/STP FCSEL easing precis FCMP FCMP FCMP	FMOV rd, rn  LDR/STR rt, [addr]  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc  tion may lead to rounding or  FCMP rn, rm  FCMN rd, #0.0  FCCMP rn, rm, #im4, cc	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm NaN  NZCV = compare(rn,rm) NZCV = compare(rn,0) If (cc) NZCV = compare(rn,rm) else NZC	(scaled address) (unscaled address) ve addresses	Υ
Note: r={D,S,H} but operands a  Between registers of equal size  Between registers and memory  unscaled address offset  Load/store pair of registers  Conditional  Note: data movement with decr  Compare  with zero  Conditional compare  Note: comparison of FP numbers	FMOV LDR/STR LDUR/STUR LDP/STP FCSEL easing precis FCMP FCMP FCMP Can lead to	FMOV rd, rn  LDR/STR rt, [addr]  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc  ion may lead to rounding or  FCMP rn, rm  FCMN rd, #0.0  FCCMP rn, rm, #im4, cc  wrong conclusions on very signs.	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm  NAN  NZCV = compare(rn,rm)  NZCV = compare(rn,0)  If (cc) NZCV = compare(rn,rm) else NZC Lmilar operands due to rounding errors	(scaled address) (unscaled address) ve addresses  V = #imm4	Υ
Note: r={D,S,H} but operands a  Between registers of equal size  Between registers and memory  unscaled address offset  Load/store pair of registers  Conditional  Note: data movement with decr  Compare  with zero  Conditional compare  Note: comparison of FP numbers  Between FP registers	FMOV LDR/STR LDUR/STUR LDP/STP FCSEL easing precis FCMP FCMP FCMP can lead to	FMOV rd, rn  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc  tion may lead to rounding or  FCMP rn, rm  FCMN rd, #0.0  FCCMP rn, rm, #im4, cc  wrong conclusions on very si	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm  NAN  NZCV = compare(rn,rm)  NZCV = compare(rn,0)  If (cc) NZCV = compare(rn,rm) else NZC Limitar operands due to rounding errors	(scaled address) (unscaled address) ve addresses $V = \#imm4$ $(r=\{D,S,H\})$	Υ
Note: r={D,S,H} but operands a Between registers of equal size Between registers and memory unscaled address offset Load/store pair of registers Conditional Note: data movement with decr  Compare with zero Conditional compare Note: comparison of FP numbers  Between FP registers signed integer to FP	FMOV LDR/STR LDUR/STUR LDP/STP FCSEL easing precis FCMP FCMP FCMP can lead to FCVT SCVTF	FMOV rd, rn  LDR/STR rt, [addr]  LDR/STR rt, [addr]  LDP rt, rm, [addr]  FCSEL rd, rn, rm, cc  tion may lead to rounding or  FCMP rn, rm  FCMN rd, #0.0  FCCMP rn, rm, #im4, cc  wrong conclusions on very si  FCVT rd, rn  SCVTF rd, rn	rt = Mem[addr]; Mem[addr] = rt rt = Mem[addr]; Mem[addr] = rt Load/store rt and rm from/to consecuti If (cc) rd = rn else rd = rm  NaN  NZCV = compare(rn,rm)  NZCV = compare(rn,0)  If (cc) NZCV = compare(rn,rm) else NZC Limitar operands due to rounding errors  rd = rn rd = rn	(scaled address) (unscaled address)  ve addresses  V = #imm4  (r={D,S,H}) (rd={D,S,H}, rn={X,W})	Υ