

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM Limited ("ARM"). **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to ARM's customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement specifically covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the ARM trademark usage guidelines <http://www.arm.com/about/trademarks/guidelines/index.php>.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20327

In this document, where the term ARM is used to refer to the company it means "ARM or any of its subsidiaries as appropriate".

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

[ISA v82A A64 xml 00bet3.1](#)[htmldiff from-](#)[\(new\)](#)[\(old\)](#)[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)[ISA v82A A64 xml 00bet3.1 OPT](#)

## A64 -- Base Instructions (alphabetic order)

[ADC](#): Add with Carry.

[ADCS](#): Add with Carry, setting flags.

[ADD \(extended register\)](#): Add (extended register).

[ADD \(immediate\)](#): Add (immediate).

[ADD \(shifted register\)](#): Add (shifted register).

[ADDS \(extended register\)](#): Add (extended register), setting flags.

[ADDS \(immediate\)](#): Add (immediate), setting flags.

[ADDS \(shifted register\)](#): Add (shifted register), setting flags.

[ADR](#): Form PC-relative address.

[ADRP](#): Form PC-relative address to 4KB page.

[AND \(immediate\)](#): Bitwise AND (immediate).

[AND \(shifted register\)](#): Bitwise AND (shifted register).

[ANDS \(immediate\)](#): Bitwise AND (immediate), setting flags.

[ANDS \(shifted register\)](#): Bitwise AND (shifted register), setting flags.

ASR (immediate): Arithmetic Shift Right (immediate): an alias of SBFM.

ASR (register): Arithmetic Shift Right (register): an alias of ASRV.

ASRV: Arithmetic Shift Right Variable.

AT: Address Translate: an alias of SYS.

[B](#): Branch.

B.cond: Branch conditionally.

BFC: Bitfield Clear, leaving other bits unchanged: an alias of BFM.

BFI: Bitfield Insert: an alias of BFM.

[BFM](#): Bitfield Move.

BFXIL: Bitfield extract and insert at low end: an alias of BFM.

[BIC \(shifted register\)](#): Bitwise Bit Clear (shifted register).

[BICS \(shifted register\)](#): Bitwise Bit Clear (shifted register), setting flags.

[BL](#): Branch with Link.

[BLR](#): Branch with Link to Register.

[BR](#): Branch to Register.

BRK: Breakpoint instruction.

CAS, CASA, CASAL, CASL: Compare and Swap word or doubleword in memory.

[CASB, CASAB, CASALB, CASLB](#): Compare and Swap byte in memory.

[CASH, CASAH, CASALH, CASLH](#): Compare and Swap halfword in memory.

[CASP, CASPA, CASPAL, CASPL](#): Compare and Swap Pair of words or doublewords in memory.

[CBNZ](#): Compare and Branch on Nonzero.

[CBZ](#): Compare and Branch on Zero.

[CCMN \(immediate\)](#): Conditional Compare Negative (immediate).

[CCMN \(register\)](#): Conditional Compare Negative (register).

[CCMP \(immediate\)](#): Conditional Compare (immediate).

[CCMP \(register\)](#): Conditional Compare (register).

CINC: Conditional Increment: an alias of CSINC.

CINV: Conditional Invert: an alias of CSINV.

CLREX: Clear Exclusive.

[CLS](#): Count leading sign bits.

[CLZ](#): Count leading zero bits.

CMN (extended register): Compare Negative (extended register): an alias of ADDS (extended register).

CMN (immediate): Compare Negative (immediate): an alias of ADDS (immediate).

CMN (shifted register): Compare Negative (shifted register): an alias of ADDS (shifted register).

CMP (extended register): Compare (extended register): an alias of SUBS (extended register).

CMP (immediate): Compare (immediate): an alias of SUBS (immediate).

CMP (shifted register): Compare (shifted register): an alias of SUBS (shifted register).

CNEG: Conditional Negate: an alias of CSNEG.

[CRC32B, CRC32H, CRC32W, CRC32X](#): CRC32 checksum.

[CRC32CB, CRC32CH, CRC32CW, CRC32CX](#): CRC32C checksum.

[CSEL](#): Conditional Select.

CSET: Conditional Set: an alias of CSINC.

CSETM: Conditional Set Mask: an alias of CSINV.

[CSINC](#): Conditional Select Increment.

[CSINV](#): Conditional Select Invert.

[CSNEG](#): Conditional Select Negation.

DC: Data Cache operation: an alias of SYS.

[DCPS1](#): Debug Change PE State to EL1..

[DCPS2](#): Debug Change PE State to EL2..

[DCPS3](#): Debug Change PE State to EL3.

[DMB](#): Data Memory Barrier.

DRPS: Debug restore process state.

[DSB](#): Data Synchronization Barrier.

[EON \(shifted register\)](#): Bitwise Exclusive OR NOT (shifted register).

[EOR \(immediate\)](#): Bitwise Exclusive OR (immediate).

[EOR \(shifted register\)](#): Bitwise Exclusive OR (shifted register).

ERET: Exception Return.

[ESB](#): Error Synchronization Barrier.

EXTR: Extract register.

[HINT](#): Hint instruction.

HLT: Halt instruction.

HVC: Hypervisor Call.

IC: Instruction Cache operation: an alias of SYS.

[ISB](#): Instruction Synchronization Barrier.

[LDADD, LDADDA, LDADDAL, LDADDL](#): Atomic add on word or doubleword in memory.

[LDADDB, LDADDAB, LDADDALB, LDADDLB](#): Atomic add on byte in memory.

[LDADDH, LDADDAH, LDADDALH, LDADDLH](#): Atomic add on halfword in memory.

[LDAR](#): Load-Acquire Register.

[LDARB](#): Load-Acquire Register Byte.

[LDARH](#): Load-Acquire Register Halfword.

[LDAXP](#): Load-Acquire Exclusive Pair of Registers.

[LDAXR](#): Load-Acquire Exclusive Register.

[LDAXRB](#): Load-Acquire Exclusive Register Byte.

[LDAXRH](#): Load-Acquire Exclusive Register Halfword.

[LDCLR, LDCLRA, LDCLRAL, LDCLRL](#): Atomic bit clear on word or doubleword in memory.

[LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB](#): Atomic bit clear on byte in memory.

[LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH](#): Atomic bit clear on halfword in memory.

[LDEOR, LDEORA, LDEORAL, LDEORL](#): Atomic exclusive OR on word or doubleword in memory.

[LDEORB, LDEORAB, LDEORALB, LDEORLB](#): Atomic exclusive OR on byte in memory.

[LDEORH, LDEORAH, LDEORALH, LDEORLH](#): Atomic exclusive OR on halfword in memory.

[LDLAR](#): Load LOAcquire Register.

[LDLARB](#): Load LOAcquire Register Byte.

[LDLARH](#): Load LOAcquire Register Halfword.

[LDNP](#): Load Pair of Registers, with non-temporal hint.

[LDP](#): Load Pair of Registers.

[LDPSW](#): Load Pair of Registers Signed Word.

[LDR \(immediate\)](#): Load Register (immediate).

LDR (literal): Load Register (literal).

[LDR \(register\)](#): Load Register (register).

[LDRB \(immediate\)](#): Load Register Byte (immediate).

[LDRB \(register\)](#): Load Register Byte (register).

[LDRH \(immediate\)](#): Load Register Halfword (immediate).

[LDRH \(register\)](#): Load Register Halfword (register).

[LDRSB \(immediate\)](#): Load Register Signed Byte (immediate).

[LDRSB \(register\)](#): Load Register Signed Byte (register).

[LDRSH \(immediate\)](#): Load Register Signed Halfword (immediate).

[LDRSH \(register\)](#): Load Register Signed Halfword (register).

[LDRSW \(immediate\)](#): Load Register Signed Word (immediate).

[LDRSW \(literal\)](#): Load Register Signed Word (literal).

[LDRSW \(register\)](#): Load Register Signed Word (register).

[LDSET, LDSETA, LDSETAL, LDSETL](#): Atomic bit set on word or doubleword in memory.

[LDSETB, LDSETAB, LDSETALB, LDSETLB](#): Atomic bit set on byte in memory.

[LDSETH, LDSETAH, LDSETALH, LDSETLH](#): Atomic bit set on halfword in memory.

[LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL](#): Atomic signed maximum on word or doubleword in memory.

[LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB](#): Atomic signed maximum on byte in memory.

[LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH](#): Atomic signed maximum on halfword in memory.

[LDSMIN, LDSMINA, LDSMINAL, LDSMINL](#): Atomic signed minimum on word or doubleword in memory.

[LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB](#): Atomic signed minimum on byte in memory.

[LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH](#): Atomic signed minimum on halfword in memory.

[LDTR](#): Load Register (unprivileged).

[LDTRB](#): Load Register Byte (unprivileged).

[LDTRH](#): Load Register Halfword (unprivileged).

[LDTRSB](#): Load Register Signed Byte (unprivileged).

[LDTRSH](#): Load Register Signed Halfword (unprivileged).

[LDTRSW](#): Load Register Signed Word (unprivileged).

[LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL](#): Atomic unsigned maximum on word or doubleword in memory.

[LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB](#): Atomic unsigned maximum on byte in memory.

[LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH](#): Atomic unsigned maximum on halfword in memory.

[LDUMIN, LDUMINA, LDUMINAL, LDUMINL](#): Atomic unsigned minimum on word or doubleword in memory.

[LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB](#): Atomic unsigned minimum on byte in memory.

[LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH](#): Atomic unsigned minimum on halfword in memory.

[LDUR](#): Load Register (unscaled).

[LDURB](#): Load Register Byte (unscaled).

[LDURH](#): Load Register Halfword (unscaled).

[LDURSB](#): Load Register Signed Byte (unscaled).

[LDURSH](#): Load Register Signed Halfword (unscaled).

[LDURSW](#): Load Register Signed Word (unscaled).

[LDXP](#): Load Exclusive Pair of Registers.

[LDXR](#): Load Exclusive Register.

[LDXRB](#): Load Exclusive Register Byte.

[LDXRH](#): Load Exclusive Register Halfword.

LSL (immediate): Logical Shift Left (immediate): an alias of UBFM.

LSL (register): Logical Shift Left (register): an alias of LSLV.

LSLV: Logical Shift Left Variable.

LSR (immediate): Logical Shift Right (immediate): an alias of UBFM.

LSR (register): Logical Shift Right (register): an alias of LSRV.

LSRV: Logical Shift Right Variable.

[MADD](#): Multiply-Add.

MNEG: Multiply-Negate: an alias of MSUB.

MOV (bitmask immediate): Move (bitmask immediate): an alias of ORR (immediate).

MOV (inverted wide immediate): Move (inverted wide immediate): an alias of MOVN.

MOV (register): Move (register): an alias of ORR (shifted register).

MOV (to/from SP): Move between register and stack pointer: an alias of ADD (immediate).

MOV (wide immediate): Move (wide immediate): an alias of MOVZ.

[MOVK](#): Move wide with keep.

[MOVN](#): Move wide with NOT.

[MOVZ](#): Move wide with zero.

[MRS](#): Move System Register.

MSR (immediate): Move immediate value to Special Register.

[MSR \(register\)](#): Move general-purpose register to System Register.

[MSUB](#): Multiply-Subtract.

MUL: Multiply: an alias of MADD.

MVN: Bitwise NOT: an alias of ORN (shifted register).

NEG (shifted register): Negate (shifted register): an alias of SUB (shifted register).

NEGS: Negate, setting flags: an alias of SUBS (shifted register).

NGC: Negate with Carry: an alias of SBC.

NGCS: Negate with Carry, setting flags: an alias of SBCS.

[NOP](#): No Operation.

[ORN \(shifted register\)](#): Bitwise OR NOT (shifted register).

[ORR \(immediate\)](#): Bitwise OR (immediate).

[ORR \(shifted register\)](#): Bitwise OR (shifted register).

[PRFM \(immediate\)](#): Prefetch Memory (immediate).

[PRFM \(literal\)](#): Prefetch Memory (literal).

[PRFM \(register\)](#): Prefetch Memory (register).

[PRFM \(unscaled offset\)](#): Prefetch Memory (unscaled offset).

[PSB CSYNC](#): Profiling Synchronization Barrier.

RBIT: Reverse Bits.

[RET](#): Return from subroutine.

[REV](#): Reverse Bytes.

[REV16](#): Reverse bytes in 16-bit halfwords.

[REV32](#): Reverse bytes in 32-bit words.

REV64: Reverse Bytes: an alias of REV.

ROR (immediate): Rotate right (immediate): an alias of EXTR.

ROR (register): Rotate Right (register): an alias of RORV.

RORV: Rotate Right Variable.

[SBC](#): Subtract with Carry.

[SBCS](#): Subtract with Carry, setting flags.

SBFIZ: Signed Bitfield Insert in Zero: an alias of SBFM.

[SBFM](#): Signed Bitfield Move.

SBFX: Signed Bitfield Extract: an alias of SBFM.

[SDIV](#): Signed Divide.

[SEV](#): Send Event.

[SEVL](#): Send Event Local.

[SMADDL](#): Signed Multiply-Add Long.

SMC: Secure Monitor Call.

SMNEGL: Signed Multiply-Negate Long: an alias of SMSUBL.

[SMSUBL](#): Signed Multiply-Subtract Long.

[SMULH](#): Signed Multiply High.

SMULL: Signed Multiply Long: an alias of SMADDL.

[STADD, STADDL](#): Atomic add on word or doubleword in memory, without return.

[STADDB, STADDLB](#): Atomic add on byte in memory, without return.

[STADDH, STADDLH](#): Atomic add on halfword in memory, without return.

[STCLR, STCLRL](#): Atomic bit clear on word or doubleword in memory, without return.

[STCLRB, STCLRLB](#): Atomic bit clear on byte in memory, without return.

[STCLRH, STCLRLH](#): Atomic bit clear on halfword in memory, without return.

[STEOR, STEORL](#): Atomic exclusive OR on word or doubleword in memory, without return.

[STEORB, STEORLB](#): Atomic exclusive OR on byte in memory, without return.

[STEORH, STEORLH](#): Atomic exclusive OR on halfword in memory, without return.

[STLLR](#): Store LORelease Register.

[STLLRB](#): Store LORelease Register Byte.

[STLLRH](#): Store LORelease Register Halfword.

[STLR](#): Store-Release Register.

[STLRB](#): Store-Release Register Byte.

[STLRH](#): Store-Release Register Halfword.

[STLXP](#): Store-Release Exclusive Pair of registers.

[STLXR](#): Store-Release Exclusive Register.

[STLXRB](#): Store-Release Exclusive Register Byte.

[STLXRH](#): Store-Release Exclusive Register Halfword.

[STNP](#): Store Pair of Registers, with non-temporal hint.

[STP](#): Store Pair of Registers.

[STR \(immediate\)](#): Store Register (immediate).

[STR \(register\)](#): Store Register (register).

[STRB \(immediate\)](#): Store Register Byte (immediate).

[STRB \(register\)](#): Store Register Byte (register).

[STRH \(immediate\)](#): Store Register Halfword (immediate).

[STRH \(register\)](#): Store Register Halfword (register).

[STSET, STSETL](#): Atomic bit set on word or doubleword in memory, without return.

[STSETB, STSETLB](#): Atomic bit set on byte in memory, without return.

[STSETH, STSETLH](#): Atomic bit set on halfword in memory, without return.

[STSMAX, STSMAXL](#): Atomic signed maximum on word or doubleword in memory, without return.

[STSMAXB, STSMAXLB](#): Atomic signed maximum on byte in memory, without return.

[STSMAXH, STSMAXLH](#): Atomic signed maximum on halfword in memory, without return.

[STSMIN, STSMINL](#): Atomic signed minimum on word or doubleword in memory, without return.

[STSMINB, STSMINLB](#): Atomic signed minimum on byte in memory, without return.

[STSMINH, STSMINLH](#): Atomic signed minimum on halfword in memory, without return.

[STTR](#): Store Register (unprivileged).

[STTRB](#): Store Register Byte (unprivileged).

[STTRH](#): Store Register Halfword (unprivileged).

[STUMAX, STUMAXL](#): Atomic unsigned maximum on word or doubleword in memory, without return.

[STUMAXB, STUMAXLB](#): Atomic unsigned maximum on byte in memory, without return.

[STUMAXH, STUMAXLH](#): Atomic unsigned maximum on halfword in memory, without return.

[STUMIN, STUMINL](#): Atomic unsigned minimum on word or doubleword in memory, without return.

[STUMINB, STUMINLB](#): Atomic unsigned minimum on byte in memory, without return.

[STUMINH, STUMINLH](#): Atomic unsigned minimum on halfword in memory, without return.



[STUR](#): Store Register (unscaled).

[STURB](#): Store Register Byte (unscaled).

[STURH](#): Store Register Halfword (unscaled).

[STXP](#): Store Exclusive Pair of registers.

[STXR](#): Store Exclusive Register.

[STXRB](#): Store Exclusive Register Byte.

[STXRH](#): Store Exclusive Register Halfword.

[SUB \(extended register\)](#): Subtract (extended register).

[SUB \(immediate\)](#): Subtract (immediate).

[SUB \(shifted register\)](#): Subtract (shifted register).

[SUBS \(extended register\)](#): Subtract (extended register), setting flags.

[SUBS \(immediate\)](#): Subtract (immediate), setting flags.

[SUBS \(shifted register\)](#): Subtract (shifted register), setting flags.

[SVC](#): Supervisor Call.

[SWP, SWPA, SWPAL, SWPL](#): Swap word or doubleword in memory.

[SWPB, SWPAB, SWPALB, SWPLB](#): Swap byte in memory.

[SWPH, SWPAH, SWPALH, SWPLH](#): Swap halfword in memory.

[SXTB](#): Signed Extend Byte: an alias of SBFM.

[SXTH](#): Sign Extend Halfword: an alias of SBFM.

[SXTW](#): Sign Extend Word: an alias of SBFM.

[SYS](#): System instruction.

[SYSL](#): System instruction with result.

[TBNZ](#): Test bit and Branch if Nonzero.

[TBZ](#): Test bit and Branch if Zero.

[TLBI](#): TLB Invalidate operation: an alias of SYS.

[TST \(immediate\)](#): Test bits (immediate): an alias of ANDS (immediate).

[TST \(shifted register\)](#): Test (shifted register): an alias of ANDS (shifted register).

[UBFIZ](#): Unsigned Bitfield Insert in Zero: an alias of UBFM.

[UBFM](#): Unsigned Bitfield Move.

[UBFX](#): Unsigned Bitfield Extract: an alias of UBFM.

[UDIV](#): Unsigned Divide.

[UMADDL](#): Unsigned Multiply-Add Long.

[UMNEGL](#): Unsigned Multiply-Negate Long: an alias of UMSUBL.

[UMSUBL](#): Unsigned Multiply-Subtract Long.

[UMULH](#): Unsigned Multiply High.

[UMULL](#): Unsigned Multiply Long: an alias of UMADDL.

UXTB: Unsigned Extend Byte: an alias of UBFM.

UXTH: Unsigned Extend Halfword: an alias of UBFM.

[WFE](#): Wait For Event.

[WFI](#): Wait For Interrupt.

[YIELD](#): YIELD.

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

<a href="#">ISA v82A A64 xml 00bet3.1</a>	<a href="#">htmldiff from-</a>	<a href="#">(new)</a>
<a href="#">(old)</a>	<a href="#">ISA_v82A_A64_xml_00bet3.1</a>	<a href="#">ISA_v82A_A64_xml_00bet3.1 OPT</a>

## A64 -- SIMD and Floating-point Instructions (alphabetic order)

ABS: Absolute value (vector).

ADD (vector): Add (vector).

ADDHN, ADDHN2: Add returning High Narrow.

ADDP (scalar): Add Pair of elements (scalar).

ADDP (vector): Add Pairwise (vector).

ADDV: Add across Vector.

[AESD](#): AES single round decryption.

[AESE](#): AES single round encryption.

[AESIMC](#): AES inverse mix columns.

[AESMC](#): AES mix columns.

[AND \(vector\)](#): Bitwise AND (vector).

[BIC \(vector, immediate\)](#): Bitwise bit Clear (vector, immediate).

[BIC \(vector, register\)](#): Bitwise bit Clear (vector, register).

[BIF](#): Bitwise Insert if False.

[BIT](#): Bitwise Insert if True.

[BSL](#): Bitwise Select.

CLS (vector): Count Leading Sign bits (vector).

CLZ (vector): Count Leading Zero bits (vector).

CMEQ (register): Compare bitwise Equal (vector).

CMEQ (zero): Compare bitwise Equal to zero (vector).

CMGE (register): Compare signed Greater than or Equal (vector).

CMGE (zero): Compare signed Greater than or Equal to zero (vector).

CMGT (register): Compare signed Greater than (vector).

CMGT (zero): Compare signed Greater than zero (vector).

CMHI (register): Compare unsigned Higher (vector).

CMHS (register): Compare unsigned Higher or Same (vector).

CMLE (zero): Compare signed Less than or Equal to zero (vector).

CMLT (zero): Compare signed Less than zero (vector).

CMTST: Compare bitwise Test bits nonzero (vector).

CNT: Population Count per byte.

DUP (element): Duplicate vector element to vector or scalar.

DUP (general): Duplicate general-purpose register to vector.

[EOR \(vector\)](#): Bitwise Exclusive OR (vector).

[EXT](#): Extract vector from pair of vectors.

FABD: Floating-point Absolute Difference (vector).

[FABS \(scalar\)](#): Floating-point Absolute value (scalar).

FABS (vector): Floating-point Absolute value (vector).

FACGE: Floating-point Absolute Compare Greater than or Equal (vector).

FACGT: Floating-point Absolute Compare Greater than (vector).

[FADD \(scalar\)](#): Floating-point Add (scalar).

FADD (vector): Floating-point Add (vector).

[FADDP \(scalar\)](#): Floating-point Add Pair of elements (scalar).

FADDP (vector): Floating-point Add Pairwise (vector).

[FCCMP](#): Floating-point Conditional quiet Compare (scalar).

[FCCMPE](#): Floating-point Conditional signaling Compare (scalar).

FCMEQ (register): Floating-point Compare Equal (vector).

FCMEQ (zero): Floating-point Compare Equal to zero (vector).

FCMGE (register): Floating-point Compare Greater than or Equal (vector).

FCMGE (zero): Floating-point Compare Greater than or Equal to zero (vector).

FCMGT (register): Floating-point Compare Greater than (vector).

FCMGT (zero): Floating-point Compare Greater than zero (vector).

FCMLE (zero): Floating-point Compare Less than or Equal to zero (vector).

FCMLT (zero): Floating-point Compare Less than zero (vector).

FCMP: Floating-point quiet Compare (scalar).

FCMPE: Floating-point signaling Compare (scalar).

FCSEL: Floating-point Conditional Select (scalar).

FCVT: Floating-point Convert precision (scalar).

[FCVTAS \(scalar\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).

FCVTAS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).

[FCVTAU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).

FCVTAU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).

FCVTL, FCVTL2: Floating-point Convert to higher precision Long (vector).

[FCVTMS \(scalar\)](#): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).

FCVTMS (vector): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).

[FCVTMU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).

FCVTMU (vector): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).

FCVTN, FCVTN2: Floating-point Convert to lower precision Narrow (vector).

[FCVTNS \(scalar\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).

FCVTNS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).

[FCVTNU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).

FCVTNU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).

[FCVTPS \(scalar\)](#): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).

FCVTPS (vector): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).

[FCVTPU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).

FCVTPU (vector): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).

FCVTXN, FCVTXN2: Floating-point Convert to lower precision Narrow, rounding to odd (vector).

[FCVTZS \(scalar, fixed-point\)](#): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).

[FCVTZS \(scalar, integer\)](#): Floating-point Convert to Signed integer, rounding toward Zero (scalar).

FCVTZS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).

FCVTZS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).

[FCVTZU \(scalar, fixed-point\)](#): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).

[FCVTZU \(scalar, integer\)](#): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).

FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).

FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).

FDIV (scalar): Floating-point Divide (scalar).

FDIV (vector): Floating-point Divide (vector).

[FMADD](#): Floating-point fused Multiply-Add (scalar).

[FMAX \(scalar\)](#): Floating-point Maximum (scalar).

FMAX (vector): Floating-point Maximum (vector).

[FMAXNM \(scalar\)](#): Floating-point Maximum Number (scalar).

FMAXNM (vector): Floating-point Maximum Number (vector).

[FMAXNMP \(scalar\)](#): Floating-point Maximum Number of Pair of elements (scalar).

FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).

[FMAXNMV](#): Floating-point Maximum Number across Vector.

[FMAXP \(scalar\)](#): Floating-point Maximum of Pair of elements (scalar).

FMAXP (vector): Floating-point Maximum Pairwise (vector).

[FMAXV](#): Floating-point Maximum across Vector.

[FMIN \(scalar\)](#): Floating-point Minimum (scalar).

FMIN (vector): Floating-point minimum (vector).

[FMINNM \(scalar\)](#): Floating-point Minimum Number (scalar).

FMINNM (vector): Floating-point Minimum Number (vector).

[FMINNMP \(scalar\)](#): Floating-point Minimum Number of Pair of elements (scalar).

FMINNMP (vector): Floating-point Minimum Number Pairwise (vector).

[FMINNMV](#): Floating-point Minimum Number across Vector.

[FMINP \(scalar\)](#): Floating-point Minimum of Pair of elements (scalar).

FMINP (vector): Floating-point Minimum Pairwise (vector).

[FMINV](#): Floating-point Minimum across Vector.

FMLA (by element): Floating-point fused Multiply-Add to accumulator (by element).

FMLA (vector): Floating-point fused Multiply-Add to accumulator (vector).

FMLS (by element): Floating-point fused Multiply-Subtract from accumulator (by element).

FMLS (vector): Floating-point fused Multiply-Subtract from accumulator (vector).

[FMOV \(general\)](#): Floating-point Move to or from general-purpose register without conversion.

[FMOV \(register\)](#): Floating-point Move register without conversion.

FMOV (scalar, immediate): Floating-point move immediate (scalar).

[FMOV \(vector, immediate\)](#): Floating-point move immediate (vector).

[FMSUB](#): Floating-point Fused Multiply-Subtract (scalar).

FMUL (by element): Floating-point Multiply (by element).

[FMUL \(scalar\)](#): Floating-point Multiply (scalar).

FMUL (vector): Floating-point Multiply (vector).

FMULX: Floating-point Multiply extended.

FMULX (by element): Floating-point Multiply extended (by element).

[FNEG \(scalar\)](#): Floating-point Negate (scalar).

FNEG (vector): Floating-point Negate (vector).

[FNMADD](#): Floating-point Negated fused Multiply-Add (scalar).

[FNMSUB](#): Floating-point Negated fused Multiply-Subtract (scalar).

[FNMUL \(scalar\)](#): Floating-point Multiply-Negate (scalar).

FRECPE: Floating-point Reciprocal Estimate.

FRECPS: Floating-point Reciprocal Step.

FRECPX: Floating-point Reciprocal exponent (scalar).

[FRINTA \(scalar\)](#): Floating-point Round to Integral, to nearest with ties to Away (scalar).

FRINTA (vector): Floating-point Round to Integral, to nearest with ties to Away (vector).

[FRINTI \(scalar\)](#): Floating-point Round to Integral, using current rounding mode (scalar).

FRINTI (vector): Floating-point Round to Integral, using current rounding mode (vector).

[FRINTM \(scalar\)](#): Floating-point Round to Integral, toward Minus infinity (scalar).

FRINTM (vector): Floating-point Round to Integral, toward Minus infinity (vector).

[FRINTN \(scalar\)](#): Floating-point Round to Integral, to nearest with ties to even (scalar).

FRINTN (vector): Floating-point Round to Integral, to nearest with ties to even (vector).

[FRINTP \(scalar\)](#): Floating-point Round to Integral, toward Plus infinity (scalar).

FRINTP (vector): Floating-point Round to Integral, toward Plus infinity (vector).

[FRINTX \(scalar\)](#): Floating-point Round to Integral exact, using current rounding mode (scalar).

FRINTX (vector): Floating-point Round to Integral exact, using current rounding mode (vector).

[FRINTZ \(scalar\)](#): Floating-point Round to Integral, toward Zero (scalar).

FRINTZ (vector): Floating-point Round to Integral, toward Zero (vector).

FRSQRT: Floating-point Reciprocal Square Root Estimate.

FRSQRTS: Floating-point Reciprocal Square Root Step.

[FSQRT \(scalar\)](#): Floating-point Square Root (scalar).

FSQRT (vector): Floating-point Square Root (vector).

[FSUB \(scalar\)](#): Floating-point Subtract (scalar).

FSUB (vector): Floating-point Subtract (vector).

INS (element): Insert vector element from another vector element.

INS (general): Insert vector element from general-purpose register.

[LD1 \(multiple structures\)](#): Load multiple single-element structures to one, two, three, or four registers.

[LD1 \(single structure\)](#): Load one single-element structure to one lane of one register.

[LD1R](#): Load one single-element structure and Replicate to all lanes (of one register).

[LD2 \(multiple structures\)](#): Load multiple 2-element structures to two registers.

[LD2 \(single structure\)](#): Load single 2-element structure to one lane of two registers.

[LD2R](#): Load single 2-element structure and Replicate to all lanes of two registers.

[LD3 \(multiple structures\)](#): Load multiple 3-element structures to three registers.

[LD3 \(single structure\)](#): Load single 3-element structure to one lane of three registers).

[LD3R](#): Load single 3-element structure and Replicate to all lanes of three registers.

[LD4 \(multiple structures\)](#): Load multiple 4-element structures to four registers.

[LD4 \(single structure\)](#): Load single 4-element structure to one lane of four registers.

[LD4R](#): Load single 4-element structure and Replicate to all lanes of four registers.

[LDNP \(SIMD&FP\)](#): Load Pair of SIMD&FP registers, with Non-temporal hint.

[LDP \(SIMD&FP\)](#): Load Pair of SIMD&FP registers.

[LDR \(immediate, SIMD&FP\)](#): Load SIMD&FP Register (immediate offset).

LDR (literal, SIMD&FP): Load SIMD&FP Register (PC-relative literal).

[LDR \(register, SIMD&FP\)](#): Load SIMD&FP Register (register offset).

[LDUR \(SIMD&FP\)](#): Load SIMD&FP Register (unscaled offset).

[MLA \(by element\)](#): Multiply-Add to accumulator (vector, by element).

[MLA \(vector\)](#): Multiply-Add to accumulator (vector).

[MLS \(by element\)](#): Multiply-Subtract from accumulator (vector, by element).

[MLS \(vector\)](#): Multiply-Subtract from accumulator (vector).

MOV (element): Move vector element to another vector element: an alias of INS (element).

MOV (from general): Move general-purpose register to a vector element: an alias of INS (general).

MOV (scalar): Move vector element to scalar: an alias of DUP (element).

MOV (to general): Move vector element to general-purpose register: an alias of UMOV.

MOV (vector): Move vector: an alias of ORR (vector, register).

MOVI: Move Immediate (vector).

[MUL \(by element\)](#): Multiply (vector, by element).

[MUL \(vector\)](#): Multiply (vector).

MVN: Bitwise NOT (vector): an alias of NOT.

[MVNI](#): Move inverted Immediate (vector).

NEG (vector): Negate (vector).

NOT: Bitwise NOT (vector).

[ORN \(vector\)](#): Bitwise inclusive OR NOT (vector).

[ORR \(vector, immediate\)](#): Bitwise inclusive OR (vector, immediate).

[ORR \(vector, register\)](#): Bitwise inclusive OR (vector, register).

[PMUL](#): Polynomial Multiply.

PMULL, PMULL2: Polynomial Multiply Long.

RADDHN, RADDHN2: Rounding Add returning High Narrow.

RBIT (vector): Reverse Bit order (vector).

[REV16 \(vector\)](#): Reverse elements in 16-bit halfwords (vector).

[REV32 \(vector\)](#): Reverse elements in 32-bit words (vector).

[REV64](#): Reverse elements in 64-bit doublewords (vector).

RSHRN, RSHRN2: Rounding Shift Right Narrow (immediate).

RSUBHN, RSUBHN2: Rounding Subtract returning High Narrow.

[SABA](#): Signed Absolute difference and Accumulate.

[SABAL, SABAL2](#): Signed Absolute difference and Accumulate Long.

[SABD](#): Signed Absolute Difference.

[SABDL, SABDL2](#): Signed Absolute Difference Long.

[SADALP](#): Signed Add and Accumulate Long Pairwise.

SADDL, SADDL2: Signed Add Long (vector).

[SADDLP](#): Signed Add Long Pairwise.

SADDLV: Signed Add Long across Vector.

SADDW, SADDW2: Signed Add Wide.

[SCVTF \(scalar, fixed-point\)](#): Signed fixed-point Convert to Floating-point (scalar).

[SCVTF \(scalar, integer\)](#): Signed integer Convert to Floating-point (scalar).

SCVTF (vector, fixed-point): Signed fixed-point Convert to Floating-point (vector).

SCVTF (vector, integer): Signed integer Convert to Floating-point (vector).

[SHA1C](#): SHA1 hash update (choose).

SHA1H: SHA1 fixed rotate.



[SHA1M](#): SHA1 hash update (majority).

[SHA1P](#): SHA1 hash update (parity).

[SHA1SU0](#): SHA1 schedule update 0.

SHA1SU1: SHA1 schedule update 1.

[SHA256H](#): SHA256 hash update (part 1).

[SHA256H2](#): SHA256 hash update (part 2).

[SHA256SU0](#): SHA256 schedule update 0.

[SHA256SU1](#): SHA256 schedule update 1.

SHADD: Signed Halving Add.

SHL: Shift Left (immediate).

SHLL, SHLL2: Shift Left Long (by element size).

SHRN, SHRN2: Shift Right Narrow (immediate).

SHSUB: Signed Halving Subtract.

SLI: Shift Left and Insert (immediate).

SMAX: Signed Maximum (vector).

SMAXP: Signed Maximum Pairwise.

SMAXV: Signed Maximum across Vector.

SMIN: Signed Minimum (vector).

SMINP: Signed Minimum Pairwise.

SMINV: Signed Minimum across Vector.

[SMLAL, SMLAL2 \(by element\)](#): Signed Multiply-Add Long (vector, by element).

[SMLAL, SMLAL2 \(vector\)](#): Signed Multiply-Add Long (vector).

[SMLSL, SMLSL2 \(by element\)](#): Signed Multiply-Subtract Long (vector, by element).

[SMLSL, SMLSL2 \(vector\)](#): Signed Multiply-Subtract Long (vector).

SMOV: Signed Move vector element to general-purpose register.

[SMULL, SMULL2 \(by element\)](#): Signed Multiply Long (vector, by element).

[SMULL, SMULL2 \(vector\)](#): Signed Multiply Long (vector).

SQABS: Signed saturating Absolute value.

SQADD: Signed saturating Add.

[SQDMLAL, SQDMLAL2 \(by element\)](#): Signed saturating Doubling Multiply-Add Long (by element).

[SQDMLAL, SQDMLAL2 \(vector\)](#): Signed saturating Doubling Multiply-Add Long.

[SQDMLSL, SQDMLSL2 \(by element\)](#): Signed saturating Doubling Multiply-Subtract Long (by element).

[SQDMLSL, SQDMLSL2 \(vector\)](#): Signed saturating Doubling Multiply-Subtract Long.

SQDMULH (by element): Signed saturating Doubling Multiply returning High half (by element).

SQDMULH (vector): Signed saturating Doubling Multiply returning High half.

[SQDMULL, SQDMULL2 \(by element\)](#): Signed saturating Doubling Multiply Long (by element).

[SQDMULL, SQDMULL2 \(vector\)](#): Signed saturating Doubling Multiply Long.

SQNEG: Signed saturating Negate.

SQRDMLAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

SQRDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

SQRDMLSH (by element): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

SQRDMLSH (vector): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

SQRDMULH (by element): Signed saturating Rounding Doubling Multiply returning High half (by element).

SQRDMULH (vector): Signed saturating Rounding Doubling Multiply returning High half.

SQRSHL: Signed saturating Rounding Shift Left (register).

SQRSHRN, SQRSHRN2: Signed saturating Rounded Shift Right Narrow (immediate).

SQRSHRUN, SQRSHRUN2: Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

SQSHL (immediate): Signed saturating Shift Left (immediate).

SQSHL (register): Signed saturating Shift Left (register).

SQSHLU: Signed saturating Shift Left Unsigned (immediate).

SQSHRN, SQSHRN2: Signed saturating Shift Right Narrow (immediate).

SQSHRUN, SQSHRUN2: Signed saturating Shift Right Unsigned Narrow (immediate).

SQSUB: Signed saturating Subtract.

SQXTN, SQXTN2: Signed saturating extract Narrow.

SQXTUN, SQXTUN2: Signed saturating extract Unsigned Narrow.

[SRHADD](#): Signed Rounding Halving Add.

SRI: Shift Right and Insert (immediate).

SRSHL: Signed Rounding Shift Left (register).

RSRSHR: Signed Rounding Shift Right (immediate).

RSRSRA: Signed Rounding Shift Right and Accumulate (immediate).

SSHL: Signed Shift Left (register).

SSHLL, SSHLL2: Signed Shift Left Long (immediate).

SSHR: Signed Shift Right (immediate).

SSRA: Signed Shift Right and Accumulate (immediate).

SSUBL, SSUBL2: Signed Subtract Long.

SSUBW, SSUBW2: Signed Subtract Wide.

[ST1 \(multiple structures\)](#): Store multiple single-element structures from one, two, three, or four registers.

[ST1 \(single structure\)](#): Store a single-element structure from one lane of one register.

[ST2 \(multiple structures\)](#): Store multiple 2-element structures from two registers.

[ST2 \(single structure\)](#): Store single 2-element structure from one lane of two registers.

[ST3 \(multiple structures\)](#): Store multiple 3-element structures from three registers.

[ST3 \(single structure\)](#): Store single 3-element structure from one lane of three registers.

[ST4 \(multiple structures\)](#): Store multiple 4-element structures from four registers.

[ST4 \(single structure\)](#): Store single 4-element structure from one lane of four registers.

[STNP \(SIMD&FP\)](#): Store Pair of SIMD&FP registers, with Non-temporal hint.

[STP \(SIMD&FP\)](#): Store Pair of SIMD&FP registers.

[STR \(immediate, SIMD&FP\)](#): Store SIMD&FP register (immediate offset).

[STR \(register, SIMD&FP\)](#): Store SIMD&FP register (register offset).

[STUR \(SIMD&FP\)](#): Store SIMD&FP register (unscaled offset).

SUB (vector): Subtract (vector).

SUBHN, SUBHN2: Subtract returning High Narrow.

SUQADD: Signed saturating Accumulate of Unsigned value.

SXTL, SXTL2: Signed extend Long: an alias of SSHLL, SSHLL2.

[TBL](#): Table vector Lookup.

[TBX](#): Table vector lookup extension.

TRN1: Transpose vectors (primary).

TRN2: Transpose vectors (secondary).

[UABA](#): Unsigned Absolute difference and Accumulate.

[UABAL, UABAL2](#): Unsigned Absolute difference and Accumulate Long.

[UABD](#): Unsigned Absolute Difference (vector).

[UABDL, UABDL2](#): Unsigned Absolute Difference Long.

[UADALP](#): Unsigned Add and Accumulate Long Pairwise.

UADDL, UADDL2: Unsigned Add Long (vector).

[UADDLP](#): Unsigned Add Long Pairwise.

UADDLV: Unsigned sum Long across Vector.

UADDW, UADDW2: Unsigned Add Wide.

[UCVTF \(scalar, fixed-point\)](#): Unsigned fixed-point Convert to Floating-point (scalar).

[UCVTF \(scalar, integer\)](#): Unsigned integer Convert to Floating-point (scalar).

UCVTF (vector, fixed-point): Unsigned fixed-point Convert to Floating-point (vector).

UCVTF (vector, integer): Unsigned integer Convert to Floating-point (vector).

UHADD: Unsigned Halving Add.

UHSUB: Unsigned Halving Subtract.

UMAX: Unsigned Maximum (vector).

UMAXP: Unsigned Maximum Pairwise.

UMAXV: Unsigned Maximum across Vector.

UMIN: Unsigned Minimum (vector).

UMINP: Unsigned Minimum Pairwise.

UMINV: Unsigned Minimum across Vector.

[UMLAL, UMLAL2 \(by element\)](#): Unsigned Multiply-Add Long (vector, by element).

[UMLAL, UMLAL2 \(vector\)](#): Unsigned Multiply-Add Long (vector).

[UMLSL, UMLSL2 \(by element\)](#): Unsigned Multiply-Subtract Long (vector, by element).

[UMLSL, UMLSL2 \(vector\)](#): Unsigned Multiply-Subtract Long (vector).

UMOV: Unsigned Move vector element to general-purpose register.

[UMULL, UMULL2 \(by element\)](#): Unsigned Multiply Long (vector, by element).

[UMULL, UMULL2 \(vector\)](#): Unsigned Multiply long (vector).

UQADD: Unsigned saturating Add.

UQRSHL: Unsigned saturating Rounding Shift Left (register).

UQRSHRN, UQRSHRN2: Unsigned saturating Rounded Shift Right Narrow (immediate).

UQSHL (immediate): Unsigned saturating Shift Left (immediate).

UQSHL (register): Unsigned saturating Shift Left (register).

UQSHRN, UQSHRN2: Unsigned saturating Shift Right Narrow (immediate).

UQSUB: Unsigned saturating Subtract.

UQXTN, UQXTN2: Unsigned saturating extract Narrow.

URECPE: Unsigned Reciprocal Estimate.

[URHADD](#): Unsigned Rounding Halving Add.

URSHL: Unsigned Rounding Shift Left (register).

URSHR: Unsigned Rounding Shift Right (immediate).

URSQRTE: Unsigned Reciprocal Square Root Estimate.

URSRA: Unsigned Rounding Shift Right and Accumulate (immediate).

USHL: Unsigned Shift Left (register).

USHLL, USHLL2: Unsigned Shift Left Long (immediate).

USHR: Unsigned Shift Right (immediate).

USQADD: Unsigned saturating Accumulate of Signed value.

USRA: Unsigned Shift Right and Accumulate (immediate).

USUBL, USUBL2: Unsigned Subtract Long.

USUBW, USUBW2: Unsigned Subtract Wide.

UXTL, UXTL2: Unsigned extend Long: an alias of USHLL, USHLL2.

UZP1: Unzip vectors (primary).

UZP2: Unzip vectors (secondary).

XTN, XTN2: Extract Narrow.

ZIP1: Zip vectors (primary).

ZIP2: Zip vectors (secondary).

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	0	0	0					Rm		0	0	0	0	0	0					Rn				Rd
op		S																													

### 32-bit (sf == 0)

ADC <Wd>, <Wn>, <Wm>

### 64-bit (sf == 1)

ADC <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzev;

(result, -) = if sub_op then
    operand2 = NOT(operand2);

(result, nzev) = AddWithCarry(operand1, operand2, PSTATE.C); (operand1, operand2, PSTATE.C);

if setflags then
    PSTATE.<N,Z,C,V> = nzev;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## ADCS

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

### 32-bit (sf == 0)

ADCS <Wd>, <Wn>, <Wm>

### 64-bit (sf == 1)

ADCS <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcvc;

(result, nzcvc) =if sub_op then
  operand2 = NOT(operand2);
(result, nzcvc) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcvc;if setflags then
  PSTATE.<N,Z,C,V> = nzcvc;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	0	1	0	1	1	0	0	1	Rm				option			imm3			Rn				Rd						
op S																															

### 32-bit (sf == 0)

```
ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit (sf == 1)

```
ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

## Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '1111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":



option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzev;
bit carry_in;

(result, -) = if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzev) = AddWithCarry(operand1, operand2, '0');
(operand1, operand2, carry_in);

if d == 31 then if setflags then
    PSTATE.<N,Z,C,V> = nzev;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias [MOV \(to/from SP\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	0	1	shift		imm12												Rn						Rd			
op S																															

### 32-bit (sf == 0)

ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}

### 64-bit (sf == 1)

ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12:imm12-Zeros(12), datasize);
  when '1x' ReservedValue();
```

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "shift":

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (to/from SP)</a>	shift == '00' && imm12 == '000000000000' && (Rd == '11111'    Rn == '11111')

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzev;
bit carry_in;

(result, -) = if sub_op then
  operand2 = NOT(operand2);
  carry_in = '1';
else
  carry_in = '0';

(result, nzev) = AddWithCarry(operand1, imm, '0');
(operand1, operand2, carry_in);

if d == 31 then if setflags then
  PSTATE.<N,Z,C,V> = nzev;

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

[\(new\)](#)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	0	1	0	1	1	shift	0						Rm																Rd
op				S																											

### 32-bit (sf == 0)

```
ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
----------	--

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzev;
bit carry_in;

(result, -) =if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzev) = AddWithCarry(operand1, operand2, '0');(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzev;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## ADDS (extended register)

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias [CMN \(extended register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	0	1	0	1	1	0	0	1	Rm				option			imm3			Rn				Rd						
op S																															

### 32-bit (sf == 0)

```
ADDS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit (sf == 1)

```
ADDS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<R>	Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m>	Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
<extend>	For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '1111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '1111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Alias Conditions

Alias	Is preferred when
<a href="#">CMN (extended register)</a>	Rd == '1111'

## Operation

```

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcvc;
bit carry_in;

(result, nzcvc) = if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcvc;

if d == 31 && !setflags then
    SP(operand1, operand2, '0');

PSTATE.<N,Z,C,V> = nzcvc; [] = result;
else
    X[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## ADDS (immediate)

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [CMN \(immediate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	1	1	0	0	0	1	shift		imm12												Rn				Rd				
op S																															

### 32-bit (sf == 0)

```
ADDS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

### 64-bit (sf == 1)

```
ADDS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12:~imm12 Zeros(12), datasize);
  when '1x' ReservedValue();
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "shift":

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

## Alias Conditions

Alias	Is preferred when
<a href="#">CMN (immediate)</a>	Rd == '11111'



## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcvc;
bit carry_in;

(result, nzcvc) =if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcvc;

if d == 31 && !setflags then
    SP(operand1, imm, '0');

PSTATE.<N,Z,C,V> = nzcvc; [] = result;
else


X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from- [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [CMN \(shifted register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	0	1	0	1	1	shift	0						Rm																Rd
op S																															

### 32-bit (sf == 0)

```
ADDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
ADDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
----------	--

## Alias Conditions

Alias	Is preferred when
<a href="#">CMN (shifted register)</a>	Rd == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X\[n\];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcvc;
bit carry_in;

(result, nzcvc) =if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, '0');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcvc;if setflags then
    PSTATE.<N,Z,C,V> = nzcvc;

X\[d\] = result;
```

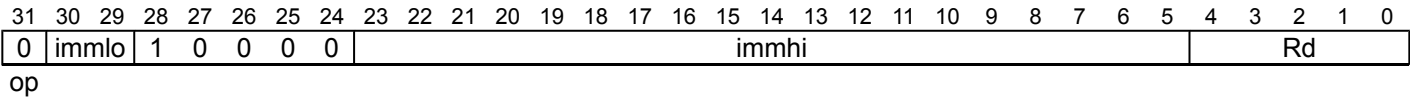
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

# ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.



## Literal

ADR <Xd>, <label>

```
integer d = UInt(Rd);
boolean page = (op == '1');
bits(64) imm;

imm = if page then
    imm = SignExtend(immhi:immlo, 64); (immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(immhi:immlo, 64);
```

## Assembler Symbols

- <Xd>                    Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <label>                Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

## Operation

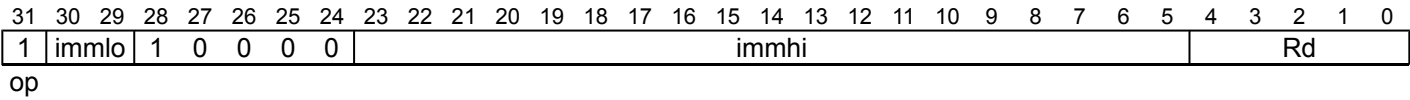
```
bits(64) base = PC[[];[]];
if page then
    base<11:0> =
Zeros(12);
X[d] = base + imm;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# ADRP

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.



## Literal

```
ADRP <Xd>, <label>
```

```
integer d = UInt(Rd);
boolean page = (op == '1');
bits(64) imm;

imm = if page then
    imm = SignExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(immhi:immlo, 64);
```

## Assembler Symbols

- <Xd>                    Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <label>                Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

## Operation

```
bits(64) base = PC[];

base<11:0> = if page then
    base<11:0> = Zeros(12);

X[d] = base + imm;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# AESD

AES single round decryption.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	0	1	1	0	Rn				Rd					
D																															

## Advanced SIMD

AESD <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
result =if decrypt then
    AESInvSubBytes(AESInvShiftRows(result));
else
    result =
    AESSubBytes(AESShiftRows(result));
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# AESE

AES single round encryption.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	Rn				Rd					
D																															

## Advanced SIMD

AESE <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
result =if decrypt then
    result = AESInvSubBytes(AESInvShiftRows(result));
else
    result = AESSubBytes(AESShiftRows(result));

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

AESIMC

AES inverse mix columns.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	1	1	1	0	Rn				Rd					
D																															

Advanced SIMD

AESIMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckCryptoEnabled64();

bits(128) operand = V[n];
bits(128) result;
result =if decrypt then
    result = AESInvMixColumns(operand);
else
    result = AESMixColumns(operand);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.



# AESMC

AES mix columns.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	1	0	1	0	Rn				Rd					
D																															

## Advanced SIMD

AESMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckCryptoEnabled64();

bits(128) operand = V[n];
bits(128) result;
result =if decrypt then
  result = AESInvMixColumns(operand);
else
  result = AESMixColumns(operand);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	Rm					0	0	0	1	1	1	Rn					Rd				
size																															

### Three registers of the same type

AND <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean invert = (size<0> == '1'); LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = operand1 AND operand2; if invert then operand2 =
NOT(operand2);

case op of
  when LogicalOp_AND
    result = operand1 AND operand2;
  when LogicalOp_ORR
    result = operand1 OR operand2;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## AND (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		0		0		1		0		0		1		0		0		N		immr						imms						Rn						Rd			
opc																																									

### 32-bit (sf == 0 && N == 0)

AND <Wd|WSP>, <Wn>, #<imm>

### 64-bit (sf == 1)

AND <Xd|SP>, <Xn>, #<imm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then boolean setflags; LogicalOp op;
case op of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;
bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

result = operand1 AND imm;
if d == 31 then case op of
  when
    LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>: IsZeroBit(result): '00';

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## AND (shifted register)

Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
sf	0	0	0	1	0	1	0	shift	0	Rm						imm6						Rn						Rd							
opc										N																									

### 32-bit (sf == 0)

AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then boolean setflags; LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);{imm6};
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 AND operand2; if invert then operand2 =
NOT(operand2);

case op of
  when LogicalOp\_AND result = operand1 AND operand2;
  when LogicalOp\_ORR result = operand1 OR operand2;
  when LogicalOp\_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## ANDS (immediate)

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [TST \(immediate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	1	0	0	1	0	0	N	immr						imms						Rn				Rd					
opc																															

### 32-bit (sf == 0 && N == 0)

```
ANDS <Wd>, <Wn>, #<imm>
```

### 64-bit (sf == 1)

```
ANDS <Xd>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

bits(datasize) imm;
if sf == '0' && N != '0' then boolean setflags; LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

## Alias Conditions

Alias	Is preferred when
<a href="#">TST (immediate)</a>	Rd == '11111'

## Operation

```
bits(datasize) result;  
bits(datasize) operand1 = X[n];  
bits(datasize) operand2 = imm;  
  
result = operand1 AND imm;  
PSTATE.<N,Z,C,V> = result<datasize-1>;case op of  
    when LogicalOp_AND result = operand1 AND operand2;  
    when LogicalOp_ORR result = operand1 OR operand2;  
    when LogicalOp_EOR result = operand1 EOR operand2;  
  
if setflags then  
    PSTATE.<N,Z,C,V> = result<datasize-1>; IsZeroBit(result):'00';  
  
if d == 31 && !setflags then  
    SP(result):'00';[] = result;  
else  
  
X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



## ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [TST \(shifted register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		1		1		0		1		0		1		0		shift		0		Rm						imm6						Rn						Rd			
opc												N																													

### 32-bit (sf == 0)

ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if sf == '0' && imm6<5> == '1' then boolean setflags; LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);{imm6};
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--

Alias Conditions

Alias	Is preferred when
<a href="#">TST (shifted register)</a>	Rd == '11111'

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>;if invert then operand2 =NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>;IsZeroBit(result):'00';

X[d] = result;
```

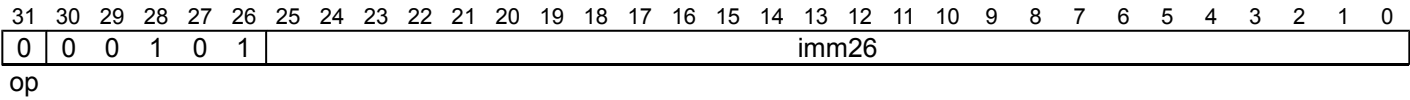
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

B

Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.



26-bit signed PC-relative branch offset

```
B <label>
```

```

BranchType branch_type = if op == '1' then BranchType_CALL else BranchType_JMP;
bits(64) offset = SignExtend(imm26:'00', 64);

```

Assembler Symbols

<code>&lt;label&gt;</code>	Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.
----------------------------	--

Operation

```

if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(PC[] + offset, BranchType_JMP);[] + offset, branch_type);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## BFM

Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits unchanged.

This instruction is used by the aliases [BFC](#), [BFI](#), and [BFXIL](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	1	1	0	0	1	1	0	N	immr						imms						Rn				Rd				
opc																															

### 32-bit (sf == 0 && N == 0)

BFM <Wd>, <Wn>, #<immr>, #<imms>

### 64-bit (sf == 1 && N == 1)

BFM <Xd>, <Xn>, #<immr>, #<imms>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then ease_opcode_of
  when '00' inzero = TRUE; extend = TRUE; // SBFM
  when '01' inzero = FALSE; extend = FALSE; // BFM
  when '10' inzero = TRUE; extend = FALSE; // UBFM
  when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt();
R = {immr};
S = UInt(immr);
{imms};
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Is preferred when
<a href="#">BFC</a>	<code>Rn == '11111' &amp;&amp; <a href="#">UInt</a>(imms) &lt; <a href="#">UInt</a>(immr)</code>
<a href="#">BFI</a>	<code>Rn != '11111' &amp;&amp; <a href="#">UInt</a>(imms) &lt; <a href="#">UInt</a>(immr)</code>
<a href="#">BFXIL</a>	<code><a href="#">UInt</a>(imms) &gt;= <a href="#">UInt</a>(immr)</code>

Operation

```
bits(datasize) dst = bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src, R) AND wmask;
(src<S>) else dst;

// combine extension bits and result bits
X[d] = (dst AND[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [\(new\)](#)  
[ISA v82A A64 xml 00bet3.1 OPT](#)

## BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
op												cmode																			

### 16-bit (cmode == 10x1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit (cmode == 0xx1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx01' operation = when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '1110x' operation = when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UnallocatedEncoding();
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff](#) from- [\(new\)](#)  
[\(old\)](#) ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	Rm					0	0	0	1	1	1	Rn					Rd				
size																															

### Three registers of the same type

BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

```
boolean invert = (size<0> == '1'); LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

operand2 =if invert then operand2 = NOT(operand2);

result = operand1 AND operand2; case op of
  when
    LogicalOp_AND
      result = operand1 AND operand2;
  when LogicalOp_ORR
      result = operand1 OR operand2;

```

V[d] = result;

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.



## BIC (shifted register)

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		0		0		0		1		0		1		0		shift		1		Rm						imm6						Rn						Rd			
opc											N																														

### 32-bit (sf == 0)

BIC <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

BIC <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then boolean setflags; LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);{imm6};
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 =if invert then operand2 = NOT(operand2);

result = operand1 AND operand2;case op of
  when
LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
sf		1		1		0		1		0		1		0		shift		1		Rm					imm6					Rn					Rd				
opc											N																												

### 32-bit (sf == 0)

BICS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

BICS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if sf == '0' && imm6<5> == '1' then boolean setflags; LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);(imm6);
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 =if invert then operand2 = NOT(operand2);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>;case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;
if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>;IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	Rm					0	0	0	1	1	1	Rn					Rd				
opc2																															

### Three registers of the same type

BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand1;  
bits(datasize) operand2;  
bits(datasize) operand3;  
bits(datasize) operand4 = V[n];
```

```
operand1 =case op of  
  when VBitOp_VEOR  
    operand1 = V[m];  
    operand2 = Zeros();  
    operand3 = Ones();  
  when VBitOp_VBSL  
    operand1 = V[m];  
    operand2 = operand1;  
    operand3 = V[d];  
  when VBitOp_VBIT  
    operand1 = V[d];  
    operand2 = operand1;  
    operand3 = V[m];  
  when VBitOp_VBIF  
    operand1 = V[d];  
operand3 = operand2 = operand1;  
operand3 = NOT(V[m]);
```

```
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);  
[d] = operand1 EOR ((operand2 EOR operand4)
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1 [\(new\)](#)  
[ISA v82A A64 xml 00bet3.1 OPT](#)

## BIT

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	0	1	Rm					0	0	0	1	1	1	Rn					Rd				
opc2																															

### Three registers of the same type

BIT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 =case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
    operand2 = operand1;
    operand3 = NOT[d];
operand3 = { V[m]; [m] };
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3); [d] = operand1 EOR ((operand2 EOR operand4)
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

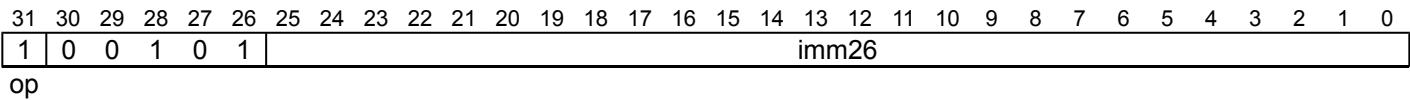
[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from- [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)



BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.



26-bit signed PC-relative branch offset

```
BL <label>
```

```

BranchType branch_type = if op == '1' then BranchType_CALL else BranchType_JMP;
bits(64) offset = SignExtend(imm26:'00', 64);

```

Assembler Symbols

<label>

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```

if branch_type == BranchType_CALL then X[30] = PC[] + 4;

BranchTo(PC[] + offset, BranchType_CALL);[] + offset, branch_type);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	Rn					0	0	0	0	0	
op																															

## Integer

BLR <Xn>

```
integer n = UInt(Rn); BranchType branch_type;
case op of
  when '00' branch_type = BranchType_JMP;
  when '01' branch_type = BranchType_CALL;
  when '10' branch_type = BranchType_RET;
  otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

## Operation

```
bits(64) target = X[n];{n};
if branch_type ==
  BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_CALL);{target, branch_type};
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	Rn				0	0	0	0	0	
op																															

## Integer

BR <Xn>

```
integer n = UInt(Rn); BranchType branch_type;
case op of
  when '00' branch_type = BranchType_JMP;
  when '01' branch_type = BranchType_CALL;
  when '10' branch_type = BranchType_RET;
  otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

## Operation

```
bits(64) target = X[n];{n};
if branch_type ==
  BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_JMP);{target, branch_type};
```

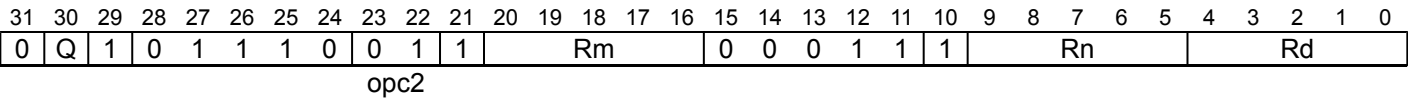
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# BSL

Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Three registers of the same type

BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "Q":
 

Q	<T>
0	8B
1	16B
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 =case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
    operand2 = operand1;
    operand3 = NOT[m];
operand3 = { V[d]; [m] };
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3); [d] = operand1 EOR ((operand2 EOR operand4)
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from- [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

### No offset (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	L	1	Rs				o0	1	1	1	1	1	Rn				Rt						

size

### Acquire (L == 1 && o0 == 0)

```
CASAB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### Acquire and release (L == 1 && o0 == 1)

```
CASALB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### No memory ordering (L == 0 && o0 == 0)

```
CASB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### Release (L == 0 && o0 == 1)

```
CASLB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if o0 == '1' then AccType ORDEREDRW else AccType ATOMICRW;
```

## Assembler Symbols

- |         |   |
|---------|---|
| <Ws>    | Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.  |
| <Wt>    | Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field. |
| <Xn SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.      |

## Operation

```
bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;
bits(8) data;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
if data == comparevalue then
[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, 1, stacctype] = newvalue;[address, datasize DIV 8, stacctype] = newvalue;

X[s] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from- [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

### No offset (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	L	1	Rs				o0	1	1	1	1	1	Rn				Rt						
size																															

### Acquire (L == 1 && o0 == 0)

CASAH <Ws>, <Wt>, [<Xn|SP>{, #0}]

### Acquire and release (L == 1 && o0 == 1)

CASALH <Ws>, <Wt>, [<Xn|SP>{, #0}]

### No memory ordering (L == 0 && o0 == 0)

CASH <Ws>, <Wt>, [<Xn|SP>{, #0}]

### Release (L == 0 && o0 == 1)

CASLH <Ws>, <Wt>, [<Xn|SP>{, #0}]

```
if !HaveAtomicExt() then UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if o0 == '1' then AccType ORDEREDRW else AccType ATOMICRW;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



## Operation

```
bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;
bits(16) data;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
if data == comparevalue then
[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, 2, stacctype] = newvalue;[address, datasize DIV 8, stacctype] = newvalue;

X[s] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

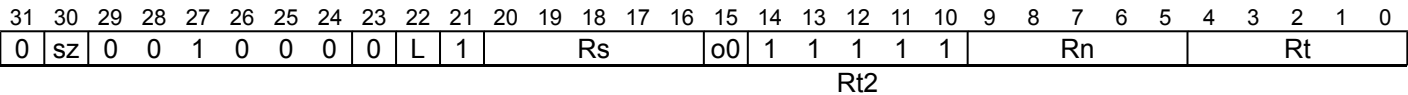
- CASPA and CASPAL load from memory with acquire semantics.
- CASPL and CASPAL store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails. If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

No offset  
(ARMv8.1)



### 32-bit, acquire (sz == 0 && L == 1 && o0 == 0)

CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]

### 32-bit, acquire and release (sz == 0 && L == 1 && o0 == 1)

CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]

### 32-bit, no memory ordering (sz == 0 && L == 0 && o0 == 0)

CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]

### 32-bit, release (sz == 0 && L == 0 && o0 == 1)

CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]

### 64-bit, acquire (sz == 1 && L == 1 && o0 == 0)

CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

### 64-bit, acquire and release (sz == 1 && L == 1 && o0 == 1)

CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

### 64-bit, no memory ordering (sz == 1 && L == 0 && o0 == 0)

CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

### 64-bit, release (sz == 1 && L == 0 && o0 == 1)

CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

```
if !HaveAtomicExt() then UnallocatedEncoding();
if Rs<0> == '1' then UnallocatedEncoding();
if Rt<0> == '1' then UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 32 << UInt(sz); {sz};
integer regsize = datasize;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field.
<W(s+1)>	Is the 32-bit name of the second general-purpose register to be compared and loaded.
<Wt>	Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field.
<W(t+1)>	Is the 32-bit name of the second general-purpose register to be conditionally stored.
<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field.
<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field.
<X(t+1)>	Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];
comparevalue = if BigEndian() then s1:s2 else s2:s1;
newvalue = if BigEndian() then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, (2*datasize) DIV 8, ldacctype];
[address, (2 * datasize) DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, (2*datasize) DIV 8, stacctype] = newvalue;
[address, (2 * datasize) DIV 8, stacctype] = newvalue;

if BigEndian() then
    X[s] = ZeroExtend(data<2*datasize-1:datasize>, datasize);{data<2*datasize-1:datasize>, regsize};
    X[s+1] = ZeroExtend(data<datasize-1:0>, datasize);
{data<datasize-1:0>, regsize};
else
    X[s] = ZeroExtend(data<datasize-1:0>, datasize);{data<datasize-1:0>, regsize};
    X[s+1] = ZeroExtend(data<2*datasize-1:datasize>, datasize);{data<2*datasize-1:datasize>, regsize};
```

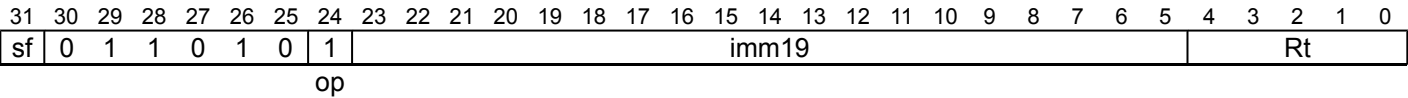
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

# CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.



32-bit (sf == 0)

```
CBNZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBNZ <Xt>, <label>
```

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

## Assembler Symbols

- <Wt>            Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <Xt>            Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <label>        Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

## Operation

```
bits(datasize) operand1 = X[t];

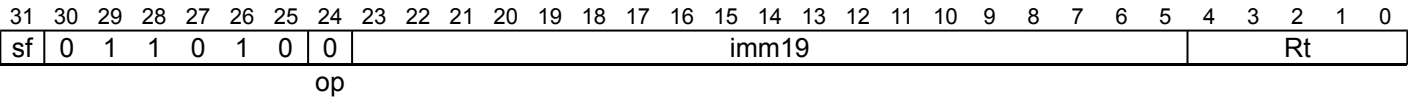
if IsZero(operand1) == FALSE then(operand1) == iszero then
  BranchTo(PC[] + offset, BranchType JMP);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



32-bit (sf == 0)

```
CBZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBZ <Xt>, <label>
```

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

## Operation

```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == TRUE then(operand1) == iszero then
  BranchTo(PC[] + offset, BranchType JMP);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## CCMN (immediate)

Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	1	0	1	0	0	1	0	imm5					cond					1	0	Rn				0	nzcw			
op																															

### 32-bit (sf == 0)

CCMN <Wn>, #<imm>, #<nzcw>, <cond>

### 64-bit (sf == 1)

CCMN <Xn>, #<imm>, #<nzcw>, <cond>

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcw;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

## Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<imm>	Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
<nzcw>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcw" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
bit carry_in = '0';

if ConditionHolds(cond) then
    (-, flags) = (condition) then
        if sub_op then
            operand2 = NOT(operand2);
            carry_in = '1';
        (-, flags) = AddWithCarry(operand1, imm, '0');
    (operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	1	0	1	0	0	1	0	Rm				cond				0	0	Rn				0	nzcw					
op																															

### 32-bit (sf == 0)

CCMN <Wn>, <Wm>, #<nzcw>, <cond>

### 64-bit (sf == 1)

CCMN <Xn>, <Xm>, #<nzcw>, <cond>

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcw;
```

## Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcw>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcw" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

if ConditionHolds(cond) then
    (-, flags) = (condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, '0');
(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.



## CCMP (immediate)

Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	1	1	1	1	0	1	0	0	1	0	imm5					cond					1	0	Rn					0	nzcw			
op																																

### 32-bit (sf == 0)

CCMP <Wn>, #<imm>, #<nzcw>, <cond>

### 64-bit (sf == 1)

CCMP <Xn>, #<imm>, #<nzcw>, <cond>

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcw;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

## Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<imm>	Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
<nzcw>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcw" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2;
bits(datasize) operand2 = imm;
bit carry_in = '0';

if ConditionHolds(cond) then
    operand2 = (condition) then
        if sub_op then
            operand2 = NOT(imm);
        (operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	1	1	0	1	0	0	1	0	Rm				cond				0	0	Rn				0	nzcw					
op																															

### 32-bit (sf == 0)

CCMP <Wn>, <Wm>, #<nzcw>, <cond>

### 64-bit (sf == 1)

CCMP <Xn>, <Xm>, #<nzcw>, <cond>

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcw;
```

## Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcw>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcw" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

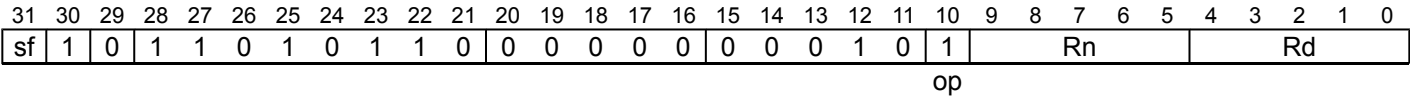
if ConditionHolds(cond) then
    operand2 = (condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

CLS

Count leading sign bits: Rd = CLS (Rn) .



32-bit (sf == 0)

CLS <Wd>, <Wn>

64-bit (sf == 1)

CLS <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32; CountOp opcode = if op == '0' then CountOp_CLZ else Cou
```

Assembler Symbols

- <Wd>                Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>                Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd>                Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>                Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
integer result;
bits(datasize) operand1 = X[n];

result =if opcode == CountOp_CLZ then
result = CountLeadingZeroBits(operand1);
else
result = CountLeadingSignBits(operand1);

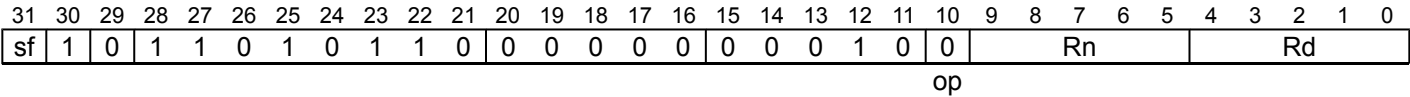
X[d] = result<datasize-1:0>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

CLZ

Count leading zero bits: Rd = CLZ (Rn).



32-bit (sf == 0)

```
CLZ <Wd>, <Wn>
```

64-bit (sf == 1)

```
CLZ <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else Cou
```

Assembler Symbols

- <Wd>                Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>                Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd>                Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>                Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
integer result;
bits(datasize) operand1 = X[n];

result =if opcode == CountOp_CLZ then
result = CountLeadingZeroBits(operand1);
else
result = CountLeadingSignBits(operand1);
X[d] = result<datasize-1:0>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction, and in ARMv8.1 it is mandatory for all implementations to implement it.

[ID\\_AA64ISAR0\\_EL1](#).CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	0	0	sz	Rn					Rd					
C																															

### CRC32B (sf == 0 && sz == 00)

CRC32B <Wd>, <Wn>, <Wm>

### CRC32H (sf == 0 && sz == 01)

CRC32H <Wd>, <Wn>, <Wm>

### CRC32W (sf == 0 && sz == 10)

CRC32W <Wd>, <Wn>, <Wm>

### CRC32X (sf == 1 && sz == 11)

CRC32X <Wd>, <Wn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding();
integer size = 8 << UInt(sz); (sz); // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
- <Wm> Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

## Operation

```
if !HaveCRCExt() then
    UnallocatedEncoding();

bits(32) acc = X[n];    // accumulator
bits(size) val = X[m]; // input value
bits(32) poly = 0x04C11DB7<31:0>;
[m]; // input value
bits(32) poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

bits(32+size) tempacc = BitReverse(acc):(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction, and in ARMv8.1 it is mandatory for all implementations to implement it.

[ID\\_AA64ISAR0\\_EL1](#).CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	0	1	sz	Rn					Rd					
C																															

### CRC32CB (sf == 0 && sz == 00)

CRC32CB <Wd>, <Wn>, <Wm>

### CRC32CH (sf == 0 && sz == 01)

CRC32CH <Wd>, <Wn>, <Wm>

### CRC32CW (sf == 0 && sz == 10)

CRC32CW <Wd>, <Wn>, <Wm>

### CRC32CX (sf == 1 && sz == 11)

CRC32CX <Wd>, <Wn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding();
integer size = 8 << UInt(sz); (sz); // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm>	Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

## Operation

```
if !HaveCRCExt() then
    UnallocatedEncoding();

bits(32) acc = X[n];    // accumulator
bits(size) val = X[m]; // input value
bits(32) poly = 0x1EDC6F41<31:0>;
[m]; // input value
bits(32) poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

bits(32+size) tempacc = BitReverse(acc):(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from- [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)



## CSEL

Conditional Select returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	0	0	Rm				cond				0	0	Rn				Rd						
op											o2																				

### 32-bit (sf == 0)

CSEL <Wd>, <Wn>, <Wm>, <cond>

### 64-bit (sf == 1)

CSEL <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    (condition) then
        result = operand1;
else
    result = operand2; result = operand2;
    if else_inv then result =
        NOT(result);
    if else_inc then result = result + 1;
X[d] = result;
```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases [CINC](#), and [CSET](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	0	0	1	1	0	1	0	1	0	0	Rm					cond					0	1	Rn					Rd				
op											o2																					

### 32-bit (sf == 0)

CSINC <Wd>, <Wn>, <Wm>, <cond>

### 64-bit (sf == 1)

CSINC <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Alias Conditions

Alias	Is preferred when
<a href="#">CINC</a>	Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm
<a href="#">CSET</a>	Rm == '11111' && cond != '111x' && Rn == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
(condition) then
    result = operand1;
else
    result = operand2 + 1; result = operand2;
if else_inv then result =
NOT(result);
if else_inc then result = result + 1;
X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## CSINV

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases [CINV](#), and [CSETM](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	1	0	1	1	0	1	0	1	0	0	Rm					cond					0	0	Rn					Rd				
op											o2																					

### 32-bit (sf == 0)

CSINV <Wd>, <Wn>, <Wm>, <cond>

### 64-bit (sf == 1)

CSINV <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Alias Conditions

Alias	Is preferred when
<a href="#">CINV</a>	Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm
<a href="#">CSETM</a>	Rm == '11111' && cond != '111x' && Rn == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
(condition) then
    result = operand1;
else
    result = result operand2;
if else_inv then result = NOT(operand2);(result);
if else_inc then result = result + 1;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias [CNEG](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	1	0	1	1	0	1	0	1	0	0	Rm					cond					0	1	Rn					Rd				
op											o2																					

### 32-bit (sf == 0)

CSNEG <Wd>, <Wn>, <Wm>, <cond>

### 64-bit (sf == 1)

CSNEG <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Alias Conditions

Alias	Is preferred when
<a href="#">CNEG</a>	cond != '111x' && Rn == Rm

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
(condition) then
    result = operand1;
else
    result = result = operand2;
if else_inv then result = NOT(operand2);
    result = result + 1; (result);
if else_inc then result = result + 1;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



DCPS1

- Debug Change PE State to EL1, when executed in Debug state:
- If executed at EL0 changes the current Exception level and SP to EL1 using SP\_EL1.
  - Otherwise, if executed at ELx, selects SP\_ELx.

- The target exception level of a DCPS1 instruction is:
- EL1 if the instruction is executed at EL0.
  - Otherwise, the Exception level at which the instruction is executed.

- When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:
- `ELR_ELx` becomes UNKNOWN.
  - `SPSR_ELx` becomes UNKNOWN.
  - `ESR_ELx` becomes UNKNOWN.
  - `DLR_EL0` and `DSPSR_EL0` become UNKNOWN.
  - The endianness is set according to `SCTLR_ELx.EE`.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and `HCR_EL2.TGE == 1`.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see [DCPS](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16																0	0	0	0	1

LL

System

```
DCPS1 {#<imm>}
```

```
if !bits(2) target_level = LL;
if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
DCPSInstruction(LL); (target_level);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

DCPS2

- Debug Change PE State to EL2, when executed in Debug state:
- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP\_EL2.
  - Otherwise, if executed at ELx, selects SP\_ELx.

- The target exception level of a DCPS2 instruction is:
- EL2 if the instruction is executed at an exception level that is not EL3.
  - EL3 if the instruction is executed at EL3.

- When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:
- *ELR\_ELx* becomes UNKNOWN.
  - *SPSR\_ELx* becomes UNKNOWN.
  - *ESR\_ELx* becomes UNKNOWN.
  - *DLR\_EL0* and *DSPSR\_EL0* become UNKNOWN.
  - The endianness is set according to *SCTLR\_ELx*.EE.

- This instruction is UNDEFINED at the following exception levels:
- All exception levels if EL2 is not implemented.
  - At EL0 and EL1 in Secure state if EL2 is implemented.

This instruction is always UNDEFINED in Non-debug state.  
For more information on the operation of the DCPSn instructions, see [DCPS](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16															0	0	0	1	0	
LL																															

System

```
DCPS2 {#<imm>}
```

```
if !bits(2) target_level = LL;
if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
DCPSInstruction(LL); (target_level);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

DCPS3

Debug Change PE State to EL3, when executed in Debug state:

- If executed at EL3 selects SP\_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP\_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

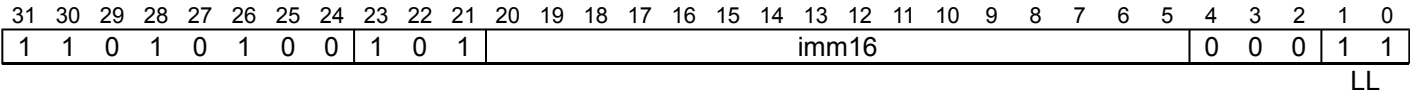
- *ELR\_EL3* becomes UNKNOWN.
- *SPSR\_EL3* becomes UNKNOWN.
- *ESR\_EL3* becomes UNKNOWN.
- *DLR\_EL0* and *DSPSR\_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR\_EL3*.EE.

This instruction is UNDEFINED at all exception levels if either:

- *EDSCR.SDD* == 1.
- EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS*.



System

```
DCPS3 {#<imm>}
```

```
if !bits(2) target_level = LL;
if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
DCPSInstruction(LL); (target_level);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see [Data Memory Barrier](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm			1	0	1	1	1	1	1	1	1
opc																															

## System

DMB <option>|<imm>

```
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
when '00' op = MemBarrierOp_DSB;
when '01' op = MemBarrierOp_DMB;
when '10' op = MemBarrierOp_ISB;
otherwise UnallocatedEncoding types;
();

case CRm<3:2> of
when '00' domain = MBReqDomain_OuterShareable;
when '01' domain = MBReqDomain_Nonshareable;
when '10' domain = MBReqDomain_InnerShareable;
when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
when '01' types = MBReqTypes_Reads;
when '10' types = MBReqTypes_Writes;
when '11' types = MBReqTypes_All;
otherwise
types = MBReqTypes_All;
domain = MBReqDomain_FullSystem;
```

## Assembler Symbols

- <option>

Specifies the limitation on the barrier operation. Values are:
- SY

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.
- ST

Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.
- LD

Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.
- ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.
- ISHST

Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

## ISHLD

Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

## NSH

Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

## NSHST

Non-shareable is the required shareability domain, writes are the required access type before the barrier instruction, and reads and writes are the required type after the barrier instruction. Encoded as CRm = 0b0110.

## NSHLD

Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

## OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

## OSHST

Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

## OSHL

Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see [Data Memory Barrier \(DMB\)](#) or see [Data Synchronization Barrier \(DSB\)](#).

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

## Operation

```
case op of
  when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier(domain, types); {};
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff](#) from- [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

**DSB**

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see [Data Synchronization Barrier](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm				1	0	0	1	1	1	1	1	1
opc																																

## System

DSB <option> | #<imm>

```
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case op of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise UnallocatedEncoding types;
());

case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
  otherwise
    types = MBReqTypes_All;
    domain = MBReqDomain_FullSystem;
```

## Assembler Symbols

<option>	Specifies the limitation on the barrier operation. Values are:
----------	--

## SY

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

## ST

Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

**LD**

Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

## ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRM = 0b1011.

## ISHST

Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

## ISHLD

Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

## NSH

Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

## NSHST

Non-shareable is the required shareability domain, writes are the required access type before the barrier instruction, and reads and writes are the required type after the barrier instruction. Encoded as CRm = 0b0110.

## NSHLD

Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

## OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

## OSHST

Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

## OSHLD

Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see [Data Memory Barrier \(DMB\)](#) or see [Data Synchronization Barrier \(DSB\)](#).

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

## Operation

```
case op of
  when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier(domain, types); {};
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff](#) from- [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
sf		1		0		0		1		0		1		0		shift		1		Rm						imm6						Rn						Rd					
opc												N																															

### 32-bit (sf == 0)

EON <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

EON <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then boolean setflags; LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);{imm6};
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--



## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 =if invert then operand2 = NOT(operand2);

result = operand1 EOR operand2;case op of
  when
LogicalOp_AND result = operand1 AND operand2;
when LogicalOp_ORR result = operand1 OR operand2;
when LogicalOp_EOR result = operand1 EOR operand2;
if setflags then
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## EOR (vector)

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	1	Rm					0	0	0	1	1	1	Rn					Rd				
opc2																															

### Three registers of the same type

EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = case op of
  when VBitOp_VEOR
    operand1 = V[m];
operand2 = Zeros();
operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
    operand2 = operand1;
    operand3 = NOT(V();[m]);
V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1 [\(new\)](#)  
[ISA v82A A64 xml 00bet3.1 OPT](#)

## EOR (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	0	0	1	0	0	N	immr						imms						Rn						Rd			
opc																															

### 32-bit (sf == 0 && N == 0)

EOR <Wd|WSP>, <Wn>, #<imm>

### 64-bit (sf == 1)

EOR <Xd|SP>, <Xn>, #<imm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then boolean setflags; LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

result = operand1 EOR imm;

if d == 31 then case op of
  when
    LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## EOR (shifted register)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
sf		1		0		0		1		0		1		0		shift		0		Rm						imm6						Rn						Rd					
opc										N																																	

### 32-bit (sf == 0)

EOR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then boolean setflags; LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);{imm6};
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 EOR operand2; if invert then operand2 =
NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## ESB

Error Synchronization Barrier is a synchronization barrier instruction to barrier between errors. This instruction can be used at all Exception levels and in Debug state. This instruction might update DISR\_EL1 and VDISR\_EL2.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the ARM(R) Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for ARMv8-A architecture profile.

### System (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	1	1	1	1
																CRm				op2											

### System

ESB

```
SystemHintOp op;
```

```
op = ifcase CRm:op2 of
```

```
  when '0000 000' op = SystemHintOp_NOP;
```

```
  when '0000 001' op = SystemHintOp_YIELD;
```

```
  when '0000 010' op = SystemHintOp_WFE;
```

```
  when '0000 011' op = SystemHintOp_WFI;
```

```
  when '0000 100' op = SystemHintOp_SEV;
```

```
  when '0000 101' op = SystemHintOp_SEVL;
```

```
  when '0010 000'
```

```
    op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP; () then SystemHintOp_ESB else
```

```
  when '0010 001'
```

```
    op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;
```

```
  otherwise op = SystemHintOp_NOP;
```



## Operation

```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();

  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, TRUE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, TRUE);
      WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, FALSE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, FALSE);
      WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBRegDomain_FullSystem, MBRegTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

otherwise // do nothing
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

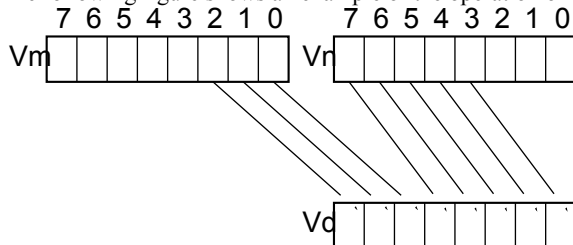
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## EXT

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for  $Q = 0$  and  $\text{imm4}<2:0> = 3$ .



Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	0	Rm				0	imm4				0	Rn				Rd						

### Advanced SIMD

EXT [<Vd>.<T>](#), [<Vn>.<T>](#), [<Vm>.<T>](#), #[<index>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if Q == '0' && imm4<3> == '1' then UnallocatedEncoding();

integer datasize = if Q == '1' then 128 else 64;
integer position = UInt(imm4) << 3;
```

### Assembler Symbols

[<Vd>](#) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

[<T>](#) Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

[<Vn>](#) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

[<Vm>](#) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

[<index>](#) Is the lowest numbered byte element to be extracted, encoded in "Q:imm4":

Q	imm4<3>	<index>
0	0	imm4<2:0>
0	1	RESERVED
1	x	imm4

## Operation

```
CheckFPAdvSIMDEnabled64\(\);  
bits(datasize) hi = V[m];  
bits(datasize) lo = V[n];  
bits\(datasize\*2\) concat = hi:lo;bits\(datasize\*2\) concat = hi:lo;  
  
V[d] = concat<position+datasize-1:position>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	0	0	1	1	0	0	0	0	Rn				Rd					
opc																															

### Half-precision (type == 11) (ARMv8.2)

FABS <Hd>, <Hn>

### Single-precision (type == 00)

FABS <Sd>, <Sn>

### Double-precision (type == 01)

FABS <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding(); FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = case fpop of
  when FPUUnaryOp_MOV result = operand;
  when FPUUnaryOp_ABS result = FPAbs(operand);
  when FPUUnaryOp_NEG result = FPNeg(operand);
  when FPUUnaryOp_SQRT result = FPSqrt(operand); (operand, FPCR);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1				Rm			0	0	1	0	1	0				Rn				Rd		
op																															

### Half-precision (type == 11) (ARMv8.2)

```
FADD <Hd>, <Hn>, <Hm>
```

### Single-precision (type == 00)

```
FADD <Sd>, <Sn>, <Sm>
```

### Double-precision (type == 01)

```
FADD <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean sub_op = (op == '1');
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) result;  
bits(datasize) operand1 = V[n];  
bits(datasize) operand2 = V[m];  
  
result =if sub_op then  
result = FPSub(operand1, operand2, FPCR);  
else  
result = FPAdd(operand1, operand2, FPCR);  
  
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	1	1	0	Rn				Rd					
SZ																															

### Half-precision

FADDP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = 32; if sz == '1' then ReservedValue();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_FADD;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	SZ	1	1	0	0	0	0	1	1	0	1	1	0	Rn				Rd					

### Single-precision and double-precision

FADDP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64; integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_FADD;
```

## Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED



For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize); ReduceOp_FADD, operand, esize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1 \(old\)](#)      [html diff from-](#)      [\(new\)](#)  
[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [ISA v82A A64 xml 00bet3.1 OPT](#)

## FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*.{N, Z, C, V} flags. If the condition does not pass then the *PSTATE*.{N, Z, C, V} flags are set to the flag bit specifier.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	Rm			cond			0	1	Rn			0	nzcv									
op																															

### Half-precision (type == 11) (ARMv8.2)

```
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
```

### Single-precision (type == 00)

```
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
```

### Double-precision (type == 01)

```
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

#### NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(cond) then
(condition) then
    flags = FPCompare(operand1, operand2, FALSE, FPCR);
(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*. {N, Z, C, V} flags. If the condition does not pass then the *PSTATE*. {N, Z, C, V} flags are set to the flag bit specifier.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1				Rm						cond		0	1				Rn		1			nzcv
																															op

### Half-precision (type == 11) (ARMv8.2)

```
FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>
```

### Single-precision (type == 00)

```
FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>
```

### Double-precision (type == 01)

```
FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

## NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

FCCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(cond) then
(condition) then
    flags = FPCompare(operand1, operand2, TRUE, FPCR);
(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	0	0	1	0	0	0	0	0	0	0	0	Rn				Rd						
rmode											opcode																				

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTAS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTAS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTAS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTAS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTAS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTAS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10' integer fltsize;
    FPConvOp op;
    FPRounding rounding;
    boolean unsigned;
    integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI();
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else;
    part = 1;
    fltsize = 64; // size of D[1] is 64
otherwise
  UnallocatedEncoding();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.



<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, (fltval, 0, unsigned, FPCR, rounding); FPRounding_TIEAWAYX);
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; {d,part} = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

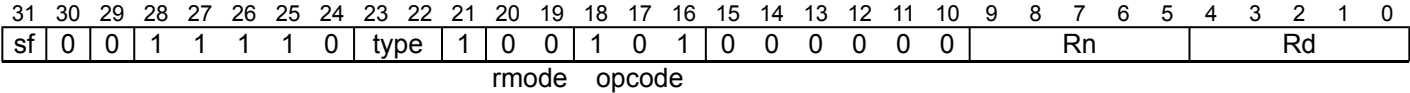
<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTAU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTAU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTAU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTAU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTAU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTAU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10' integer fltsize;
    FPConvOp op;
    FPRounding rounding;
    boolean unsigned;
    integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI();
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else;
    part = 1;
    fltsize = 64; // size of D[1] is 64
otherwise
  UnallocatedEncoding();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, (fltval, 0, unsigned, FPCR, rounding), FPRounding_TIEAWAY);
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; {d,part} = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	1	0	0	0	0	0	0	0	0	0	0	0	Rn				Rd					
											rmode		opcode																		

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTMS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTMS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTMS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTMS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTMS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTMS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(rmode); ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding); (fltval, 0, unsigned, FPCR, rounding);
  X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; (d,part) = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	1	0	0	0	1	0	0	0	0	0	0	Rn				Rd						
											rmode		opcode																		

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTMU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTMU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTMU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTMU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTMU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTMU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(rmode); ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding); (fltval, 0, unsigned, FPCR, rounding);
  X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; (d,part) = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

### FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn				Rd					
											rmode		opcode																		

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTNS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTNS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTNS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTNS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTNS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTNS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(rmode); ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding); (fltval, 0, unsigned, FPCR, rounding);
  X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; (d,part) = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	0	0	0	0	1	0	0	0	0	0	0	Rn				Rd						
rmode											opcode																				

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTNU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTNU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTNU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTNU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTNU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTNU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(rmode); ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding); (fltval, 0, unsigned, FPCR, rounding);
  X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; (d,part) = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	0	1	0	0	0	0	0	0	0	0	0	0	Rn				Rd					
rmode											opcode																				

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTPS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTPS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTPS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTPS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTPS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTPS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(rmode); ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPCConvOp_CVT_FtoI
    fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding); (fltval, 0, unsigned, FPCR, rounding);
X[d] = intval;
  when FPCConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPCConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPCConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; {d,part} = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	0	1	0	0	1	0	0	0	0	0	0	Rn				Rd						
rmode											opcode																				

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTPU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTPU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTPU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTPU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTPU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTPU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(rmode); ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding); (fltval, 0, unsigned, FPCR, rounding);
  X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; (d,part) = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	0	1	1	0	0	0	scale				Rn				Rd								
rmode										opcode																					

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

```
FCVTZS <Wd>, <Hn>, #<fbits>
```

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

```
FCVTZS <Xd>, <Hn>, #<fbits>
```

**Single-precision to 32-bit (sf == 0 && type == 00)**

```
FCVTZS <Wd>, <Sn>, #<fbits>
```

**Single-precision to 64-bit (sf == 1 && type == 00)**

```
FCVTZS <Xd>, <Sn>, #<fbits>
```

**Double-precision to 32-bit (sf == 0 && type == 01)**

```
FCVTZS <Wd>, <Dn>, #<fbits>
```

**Double-precision to 64-bit (sf == 1 && type == 01)**

```
FCVTZS <Xd>, <Dn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' integer fltsize; FPCnvOp op;
  FPRounding rounding;
  boolean unsigned;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
  when '00 11' // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPCnvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPCnvOp_CVT_ItoF;
  otherwise
    UnallocatedEncoding(scale); ();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	<p>For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".</p> <p>For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".</p>

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval =case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
intval = FPToFixed(fltval, fracbits, FALSE, FPCR, (fltval, fracbits, unsigned, FPCR, rounding); FPRoundi
  when
    FPConvOp_CVT_ItoF
      intval = X[n];
      fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
      V[d] = intval;[d] = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	1	1	0	0	0	0	0	0	0	0	0	0	Rn				Rd					
rmode										opcode																					

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTZS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTZS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTZS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTZS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTZS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTZS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(rmode); ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPCConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding); (fltval, 0, unsigned, FPCR, rounding);
  X[d] = intval;
  when FPCConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPCConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPCConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; {d,part} = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	0	1	1	0	0	1	scale				Rn				Rd								
rmode										opcode																					

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

```
FCVTZU <Wd>, <Hn>, #<fbits>
```

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

```
FCVTZU <Xd>, <Hn>, #<fbits>
```

**Single-precision to 32-bit (sf == 0 && type == 00)**

```
FCVTZU <Wd>, <Sn>, #<fbits>
```

**Single-precision to 64-bit (sf == 1 && type == 00)**

```
FCVTZU <Xd>, <Sn>, #<fbits>
```

**Double-precision to 32-bit (sf == 0 && type == 01)**

```
FCVTZU <Wd>, <Dn>, #<fbits>
```

**Double-precision to 64-bit (sf == 1 && type == 01)**

```
FCVTZU <Xd>, <Dn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' integer fltsize; FPConvOp op;
  FPRounding rounding;
  boolean unsigned;

  case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UnallocatedEncoding();
    when '11'
      if HaveFP16Ext() then
        fltsize = 16;
      else
        UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
  when '00 11' // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  otherwise
    UnallocatedEncoding(scale); ();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".  For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval =case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
intval = FPToFixed(fltval, fracbits, TRUE, FPCR, {fltval, fracbits, unsigned, FPCR, rounding}; FPRounding
  when
    FPConvOp_CVT_ItoF
      intval = X[n];
      fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
      V[d] = intval;[d] = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	1	1	0	0	1	0	0	0	0	0	0	Rn				Rd						
rmode												opcode																			



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTZU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTZU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTZU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTZU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTZU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTZU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(rmode); ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding); (fltval, 0, unsigned, FPCR, rounding);
  X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = intval; (d,part) = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type	0	Rm				0	Ra				Rn				Rd								
										o1										o0											

### Half-precision (type == 11)

(ARMv8.2)

FMADD <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (type == 00)

FMADD <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (type == 01)

FMADD <Dd>, <Dn>, <Dm>, <Da>

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result =if opa_neg then operanda = FPNeg(operanda);
if opl_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1				Rm			0	1	0	0	1	0				Rn				Rd		
op																															

### Half-precision (type == 11)

(ARMv8.2)

FMAX <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FMAX <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FMAX <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding(); FPMaXMinOp operation;
case op of
  when '00' operation = FPMaXMinOp_MAX;
  when '01' operation = FPMaXMinOp_MIN;
  when '10' operation = FPMaXMinOp_MAXNUM;
  when '11' operation = FPMaXMinOp_MINNUM;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMinNum(operand1, operand2, FPCR);
V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1					Rm		0	1	1	0	1	0				Rn				Rd		
op																															

### Half-precision (type == 11) (ARMv8.2)

FMAXNM <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FMAXNM <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FMAXNM <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding(); FPMaXMinOp operation;
case op of
  when '00' operation = FPMaXMinOp_MAX;
  when '01' operation = FPMaXMinOp_MIN;
  when '10' operation = FPMaXMinOp_MAXNUM;
  when '11' operation = FPMaXMinOp_MINNUM;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.



- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result =case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMinNum(operand1, operand2, FPCR);
V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1								sz																							

### Half-precision

FMAXNMP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32; if sz == '1' then ReservedValue();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMAXNMP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64; integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

## Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2S
1	2D

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize); ReduceOp_FMAXNUM, operand, esize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff](#) from- [\(new\)](#)  
[\(old\)](#) ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

### Half-precision

FMAXNMV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FM
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMAXNMV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
if sz:Q != '01' then ReservedValue(); // .4S only
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FM
```

### Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<b>Q</b>	<b>&lt;T&gt;</b>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<b>Q</b>	<b>sz</b>	<b>&lt;T&gt;</b>
0	x	RESERVED
1	0	4S
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize); ReduceOp_FMAXNUM, operand, esize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1 \(old\)](#)      [htmldiff from-](#)      [\(new\)](#)  
[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [ISA v82A A64 xml 00bet3.1 OPT](#)

## FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
o1								sz																							

### Half-precision

FMAXP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32; if sz == '1' then ReservedValue();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMAXP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64; integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

## Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2S
1	2D

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize); ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

### Half-precision

FMAXV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMAXV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
if sz:Q != '01' then ReservedValue();
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

## Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED



- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize);ReduceOp_FMAX, operand, esize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1				Rm			0	1	0	1	1	0			Rn					Rd		
op																															

### Half-precision (type == 11)

(ARMv8.2)

FMIN <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FMIN <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FMIN <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding(); FPMaXMinOp operation;
case op of
  when '00' operation = FPMaXMinOp_MAX;
  when '01' operation = FPMaXMinOp_MIN;
  when '10' operation = FPMaXMinOp_MAXNUM;
  when '11' operation = FPMaXMinOp_MINNUM;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result =case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMIn(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMInNum(operand1, operand2, FPCR);
V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## FMINNM (scalar)

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1					Rm		0	1	1	1	0				Rn					Rd		
op																															

### Half-precision (type == 11) (ARMv8.2)

FMINNM <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FMINNM <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FMINNM <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding(); FPMaXMinOp operation;
case op of
  when '00' operation = FPMaXMinOp_MAX;
  when '01' operation = FPMaXMinOp_MIN;
  when '10' operation = FPMaXMinOp_MAXNUM;
  when '11' operation = FPMaXMinOp_MINNUM;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
o1 sz																															

### Half-precision

FMINNMP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32; if sz == '1' then ReservedValue();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMINNMP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64; integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

## Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2S
1	2D

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize); ReduceOp_FMINNUM, operand, esize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## FMINNMV

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn						Rd					
o1																																	

### Half-precision

FMINNMV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMINNUM;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMINNMV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
if sz:Q != '01' then ReservedValue(); // .4S only
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMINNUM;
```

### Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:



<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<b>Q</b>	<b>&lt;T&gt;</b>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<b>Q</b>	<b>sz</b>	<b>&lt;T&gt;</b>
0	x	RESERVED
1	0	4S
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize); ReduceOp_FMINNUM, operand, esize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1 \(old\)](#)      [htmldiff from-](#)      [\(new\)](#)  
[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [ISA v82A A64 xml 00bet3.1 OPT](#)

## FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
o1 sz																															

### Half-precision

FMINP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32; if sz == '1' then ReservedValue();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMINP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64; integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

## Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2S
1	2D

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize); ReduceOp_FMIN, operand, esize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff](#) from- [\(new\)](#)  
[\(old\)](#) ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0										
o1																Rn								Rd							

### Half-precision

FMINV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UnallocatedEncoding();
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0										
o1																Rn								Rd							

### Single-precision and double-precision

FMINV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
if sz:Q != '01' then ReservedValue();
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

## Assembler Symbols

<V>

For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce((op, operand, esize);ReduceOp_FMIN, operand, esize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htldiff from-  
 ISA\_v82A\_A64\_xml\_00bet3.1

[\(new\)](#)  
[ISA v82A A64 xml 00bet3.1 OPT](#)

## FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	1	1	d	e	f	g	h	Rd				

### Half-precision

FMOV <Vd>.<T>, #<imm>

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;

imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:(imm8<6>, 2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	0	1	d	e	f	g	h	Rd				
cmode																															

Single-precision (op == 0)

```
FMOV <Vd>.<T>, #<imm>
```

Double-precision (Q == 1 && op == 1)

```
FMOV <Vd>.2D, #<imm>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then bits(64) imm64; ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI();
?

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

Operation

```
CheckFPAdvSIMDEnabled64();

V[rd] = imm;
```

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)



## FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	0	0	0	1	0	0	0	0	Rn				Rd					
opc																															

### Half-precision (type == 11) (ARMv8.2)

FMOV <Hd>, <Hn>

### Single-precision (type == 00)

FMOV <Sd>, <Sn>

### Double-precision (type == 01)

FMOV <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding(); FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];[n];

case fpop of
  when

FPUnaryOp_MOV result = operand;
  when FPUnaryOp_ABS result = FPAbs(operand);
  when FPUnaryOp_NEG result = FPNeg(operand);
  when FPUnaryOp_SQRT result = FPSqrt(operand, FPCR);

V[d] = operand;[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	0	x	1	1	x	0	0	0	0	0	0	Rn				Rd						
rmode												opcode																			

**Half-precision to 32-bit (sf == 0 && type == 11 && rmode == 00 && opcode == 110)**  
(ARMv8.2)

FMOV <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11 && rmode == 00 && opcode == 110)**  
(ARMv8.2)

FMOV <Xd>, <Hn>

**32-bit to half-precision (sf == 0 && type == 11 && rmode == 00 && opcode == 111)**  
(ARMv8.2)

FMOV <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && type == 00 && rmode == 00 && opcode == 111)**

FMOV <Sd>, <Wn>

**Single-precision to 32-bit (sf == 0 && type == 00 && rmode == 00 && opcode == 110)**

FMOV <Wd>, <Sn>

**64-bit to half-precision (sf == 1 && type == 11 && rmode == 00 && opcode == 111)**  
(ARMv8.2)

FMOV <Hd>, <Xn>

**64-bit to double-precision (sf == 1 && type == 01 && rmode == 00 && opcode == 111)**

FMOV <Dd>, <Xn>

**64-bit to top half of 128-bit (sf == 1 && type == 10 && rmode == 01 && opcode == 111)**

FMOV <Vd>.D[1], <Xn>

**Double-precision to 64-bit (sf == 1 && type == 01 && rmode == 00 && opcode == 110)**

FMOV <Xd>, <Dn>

**Top half of 128-bit to 64-bit (sf == 1 && type == 10 && rmode == 01 && opcode == 110)**

FMOV <Xd>, <Vn>.D[1]

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp\_CVT\_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp\_CVT\_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding\_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp\_CVT\_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp\_MOV\_ItoF else FPConvOp\_MOV\_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp\_MOV\_ItoF else FPConvOp\_MOV\_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding();

```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n, part];
    [n, part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d, part] = fltval; [d, part] = fltval;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type	0	Rm				1	Ra				Rn				Rd								
										o1										o0											

### Half-precision (type == 11)

(ARMv8.2)

FMSUB <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (type == 00)

FMSUB <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (type == 01)

FMSUB <Dd>, <Dn>, <Dm>, <Da>

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operand1 =if opa_neg then operanda = FPNeg(operand1);
result =(operanda);
if opl_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>



## FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	1	1	1	1	0	type	1	Rm						0	0	0	0	1	0	Rn						Rd					
op																																	

### Half-precision (type == 11) (ARMv8.2)

FMUL <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FMUL <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FMUL <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean negated = (op == '1');
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR); {operand1, operand2, FPCR};
if negated then result =
FPNeg(result);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	1	0	1	0	0	0	0	Rn						Rd				
opc																															

### Half-precision (type == 11)

(ARMv8.2)

FNEG <Hd>, <Hn>

### Single-precision (type == 00)

FNEG <Sd>, <Sn>

### Double-precision (type == 01)

FNEG <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding(); FPUnaryOp fpop;
case opc of
  when '00' fpop = FPUnaryOp_MOV;
  when '01' fpop = FPUnaryOp_ABS;
  when '10' fpop = FPUnaryOp_NEG;
  when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = case fpop of
  when FPUUnaryOp_MOV result = operand;
  when FPUUnaryOp_ABS result = FPAbs(operand);
  when FPUUnaryOp_NEG result = FPNeg(operand);
  when FPUUnaryOp_SQRT result = FPSqrt(operand); (operand, FPCR);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type	1				Rm			0			Ra					Rn					Rd		
										o1											o0										

### Half-precision (type == 11)

(ARMv8.2)

FNMADD <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (type == 00)

FNMADD <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (type == 01)

FNMADD <Dd>, <Dn>, <Dm>, <Da>

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operanda =if opa_neg then operanda = FPNeg(operanda);
operand1 =if opl_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type	1	Rm				1	Ra				Rn				Rd								
										o1										o0											

### Half-precision (type == 11)

(ARMv8.2)

FNMSUB <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (type == 00)

FNMSUB <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (type == 01)

FNMSUB <Dd>, <Dn>, <Dm>, <Da>

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operanda =if opa_neg then operanda = FPNeg(operanda);
result =if opl_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA_v82A_A64_xml 00bet3.1 OPT</u>



## FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1				Rm			1	0	0	0	1	0				Rn				Rd		
op																															

### Half-precision (type == 11) (ARMv8.2)

```
FNMUL <Hd>, <Hn>, <Hm>
```

### Single-precision (type == 00)

```
FNMUL <Sd>, <Sn>, <Sm>
```

### Double-precision (type == 01)

```
FNMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean negated = (op == '1');
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) result;  
bits(datasize) operand1 = V[n];  
bits(datasize) operand2 = V[m];  
  
result = FPMul(operand1, operand2, FPCR);  
  
result =if negated then result = FPNeg(result);  
  
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	1	0	0	1	0	0	0	0	Rn					Rd					
rmode																															

### Half-precision (type == 11) (ARMv8.2)

```
FRINTA <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTA <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTA <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();

boolean exact = FALSE; FPRounding rounding;
case rmode of
  when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

- <Sd>Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn>Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, (operand, FPCR, rounding, exact); FPRounding_TIEAWAY, FALSE);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	1	1	1	0	0	0	0	Rn					Rd						
rmode																															

### Half-precision (type == 11) (ARMv8.2)

```
FRINTI <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTI <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTI <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Hn>Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd>Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn>Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	0	1	0	0	0	0	0	Rn						Rd					
rmode																															

### Half-precision (type == 11) (ARMv8.2)

```
FRINTM <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTM <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTM <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPDecodeRounding('10'); {rmode<1:0>};
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64() ;

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE); (operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



## FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	0	0	0	1	0	0	0	0	Rn					Rd					
rmode																															

### Half-precision (type == 11) (ARMv8.2)

FRINTN <Hd>, <Hn>

### Single-precision (type == 00)

FRINTN <Sd>, <Sn>

### Double-precision (type == 01)

FRINTN <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPDecodeRounding('00'); {rmode<1:0>};
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64() ;

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE); (operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	0	0	1	1	0	0	0	0											
rmode																Rn				Rd											

### Half-precision (type == 11) (ARMv8.2)

```
FRINTP <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTP <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTP <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPDecodeRounding('01'); {rmode<1:0>};
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```

CheckFPAdvSIMDEnabled64 ();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from-  
 ISA\_v82A\_A64\_xml\_00bet3.1

[\(new\)](#)  
[ISA v82A A64 xml 00bet3.1 OPT](#)

## FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

An Inexact exception is raised when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	1	1	0	1	0	0	0	0	Rn					Rd					
rmode																															

### Half-precision (type == 11)

(ARMv8.2)

```
FRINTX <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTX <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTX <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```

CheckFPAdvSIMDEnabled64() ;

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, TRUE);(operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	0	1	1	1	0	0	0	0					Rn					Rd	
																rmode															

### Half-precision (type == 11) (ARMv8.2)

```
FRINTZ <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTZ <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTZ <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPDecodeRounding('11'); {rmode<1:0>};
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64() ;

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE); (operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a> <a href="#">(old)</a>	htmlldiff from-	<a href="#">(new)</a> <a href="#">ISA_v82A_A64_xml_00bet3.1</a>
		<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



## FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	1	1	1	0	0	0	0	Rn				Rd						
opc																															

### Half-precision (type == 11)

(ARMv8.2)

FSQRT <Hd>, <Hn>

### Single-precision (type == 00)

FSQRT <Sd>, <Sn>

### Double-precision (type == 01)

FSQRT <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
case opc of
  when '00' fpop = FPUnaryOp_MOV;
  when '01' fpop = FPUnaryOp_ABS;
  when '10' fpop = FPUnaryOp_NEG;
  when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = case fpop of
  when FPUUnaryOp_MOV result = operand;
  when FPUUnaryOp_ABS result = FPAbs(operand);
  when FPUUnaryOp_NEG result = FPNeg(operand);
  when FPUUnaryOp_SQRT result = FPSqrt(operand, FPCR);

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	Rm				0	0	1	1	1	0	Rn				Rd				op			

### Half-precision (type == 11)

(ARMv8.2)

FSUB <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FSUB <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FSUB <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding();
boolean sub_op = (op == '1');
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result =if sub_op then
    result = FPSub(operand1, operand2, FPCR);
else
    result =
    FPAdd(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are unallocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm			op2			1	1	1	1	1	

### Hints 6 and 7 (CRm == 0000 && op2 == 11x)

HINT #<imm>

### Hints 8 to 15, and 24 to 127 (CRm != 00x0)

HINT #<imm>

### Hints 17 to 23 (CRm == 0010 && op2 != 00x)

HINT #<imm>

```
SystemHintOp_op;  
case CRm:op2 of  
  when '0000 000' op = SystemHintOp_NOP;  
  when '0000 001' op = SystemHintOp_YIELD;  
  when '0000 010' op = SystemHintOp_WFE;  
  when '0000 011' op = SystemHintOp_WFI;  
  when '0000 100' op = SystemHintOp_SEV;  
  when '0000 101' op = SystemHintOp_SEVL;  
  when '0010 000'  
    op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;  
  when '0010 001'  
    op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;  
  otherwise op = SystemHintOp_NOP;
```

## Assembler Symbols

<imm> Is a 7-bit unsigned immediate, in the range 0 to 127, excluding the allocated encodings described below, encoded in "CRm:op2".

The following encodings of "CRm:op2" are allocated:

0000000

NOP

0000001

YIELD

0000010

WFE

0000011

WFI

0000100

SEV

0000101

SEVL

For allocated encodings of "CRm:op2":

- A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
- An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.

## Operation

```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();

  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, TRUE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, TRUE);
      WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, FALSE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, FALSE);
      WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

  otherwise // do nothing
```

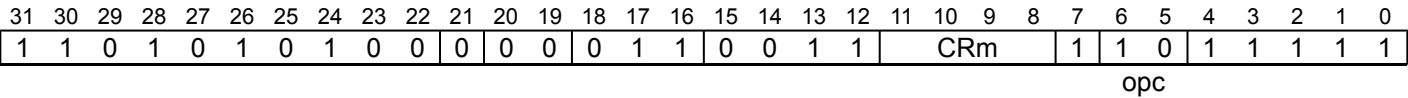
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

ISA v82A A64 xml 00bet3.1 (old)      htldiff from- (new)      ISA\_v82A\_A64\_xml\_00bet3.1      ISA v82A A64 xml 00bet3.1 OPT

# ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see [Instruction Synchronization Barrier \(ISB\)](#).



## System

ISB {<option>|<imm>}

```

MemBarrierOp op;
MBRegDomain domain;
MBRegTypes types;

case op of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise UnallocatedEncoding();

case CRm<3:2> of
  when '00' domain = MBRegDomain_OuterShareable;
  when '01' domain = MBRegDomain_Nonshareable;
  when '10' domain = MBRegDomain_InnerShareable;
  when '11' domain = MBRegDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBRegTypes_Reads;
  when '10' types = MBRegTypes_Writes;
  when '11' types = MBRegTypes_All;
  otherwise
    types = MBRegTypes_All;
    domain = MBRegDomain_FullSystem// Empty;
  
```

## Assembler Symbols

- <option>

**SY**

<imm>

Specifies an optional limitation on the barrier operation. Values are:

Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

## Operation

```

case op of
  when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	x	x	1	x	size											
L										opcode																					

### One register (opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>]
```

### Two registers (opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

### Three registers (opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

### Four registers (opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = integer UNKNOWN;  
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm				x	x	1	x	size		Rn				Rt						
L											opcode																				



One register, immediate offset (Rm == 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

- <Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	0	S	size	Rn			Rt							
L										R	opcode																				

### 8-bit (opcode == 000)

```
LD1 { <Vt>.B } [<index>], [<Xn|SP>]
```

### 16-bit (opcode == 010 && size == x0)

```
LD1 { <Vt>.H } [<index>], [<Xn|SP>]
```

### 32-bit (opcode == 100 && size == 00)

```
LD1 { <Vt>.S } [<index>], [<Xn|SP>]
```

### 64-bit (opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D } [<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm				x	x	0	S	size		Rn				Rt						
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD1 { <Vt>.B } [<index>], [<Xn|SP>], #1
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD1 { <Vt>.B } [<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD1 { <Vt>.H } [<index>], [<Xn|SP>], #2
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD1 { <Vt>.H } [<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD1 { <Vt>.S } [<index>], [<Xn|SP>], #4
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD1 { <Vt>.S } [<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D } [<index>], [<Xn|SP>], #8
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D } [<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<I>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	size											
L R										opcode S																		Rt			

### No offset

```
LD1R { <Vt>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					1	1	0	0	size	Rn					Rt					
L R										opcode S																					

### Immediate offset (Rm == 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D



<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "size":

size	<imm>
00	#1
01	#2
10	#4
11	#8

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q); // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see [LD3 \(multiple structures\)](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	size	Rn					Rt					
L										opcode																					

### No offset

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm					1	0	0	0	size	Rn					Rt					
L										opcode																					

### Immediate offset (Rm == 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#16
1	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType\_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType\_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LD2 (single structure)

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	0	S	size	Rn			Rt							
L										R	opcode																				

### 8-bit (opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B } [<index>], [<Xn|SP>]
```

### 16-bit (opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H } [<index>], [<Xn|SP>]
```

### 32-bit (opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S } [<index>], [<Xn|SP>]
```

### 64-bit (opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D } [<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = integer UNKNOWN;  
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm				x	x	0	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B } [<index>], [<Xn|SP>], #2
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B } [<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H } [<index>], [<Xn|SP>], #4
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H } [<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S } [<index>], [<Xn|SP>], #8
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S } [<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D } [<index>], [<Xn|SP>], #16
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D } [<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	size	Rn				Rt						
L R										opcode S																					

### No offset

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = integer UNKNOWN;  
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm				1	1	0	0	size	Rn				Rt							
L R										opcode S																					

### Immediate offset (Rm == 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = UInt(Rm);  
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "size":

size	<imm>
00	#2
01	#4
10	#8
11	#16

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

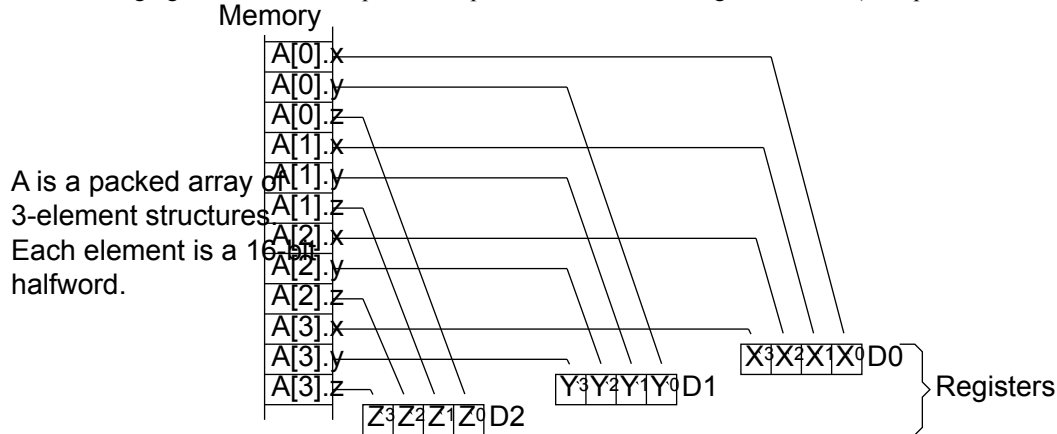
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:



Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	size	Rn					Rt					
L										opcode																					

### No offset

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm				0	1	0	0	size	Rn				Rt							
L										opcode																					

### Immediate offset (Rm == 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

- <Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T>

Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm>

Is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#24
1	#48

- <Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD3 (single structure)

Load single 3-element structure to one lane of three registers). This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	1	S	size	Rn			Rt							
L R										opcode																					

### 8-bit (opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>]
```

### 16-bit (opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>]
```

### 32-bit (opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>]
```

### 64-bit (opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = integer UNKNOWN;  
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					x	x	1	S	size	Rn					Rt					
L										R	opcode																				



### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>], #3
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>], #6
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>], #12
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>], #24
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	0	size	Rn					Rt					
L										R	opcode										S										

### No offset

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = integer UNKNOWN;  
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					1	1	1	0	size	Rn					Rt					
L										R	opcode					S															

### Immediate offset (Rm == 11111)

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = UInt(Rm);  
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "size":

size	<imm>
00	#3
01	#6
10	#12
11	#24

- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see [LD3 \(multiple structures\)](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	size	Rn				Rt						
L										opcode																					

### No offset

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm				0	0	0	0	size	Rn				Rt							
L										opcode																					

### Immediate offset (Rm == 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

### Register offset (Rm != 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();

```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType\_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType\_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	1	S	size	Rn				Rt						
L										R	opcode																				

### 8-bit (opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

### 16-bit (opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

### 32-bit (opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

### 64-bit (opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm				x	x	1	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	1	0	size	Rn				Rt						
L R										opcode S																					

### No offset

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm				1	1	1	0	size	Rn				Rt							
L										R	opcode										S										

### Immediate offset (Rm == 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "size":

size	<imm>
00	#4
01	#8
10	#16
11	#32

- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



# LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- LDADD has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).  
 For information about memory accesses see [Load/Store addressing modes](#).

## Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	0	0	0	0	0	Rn				Rt						
size				V				o3								opc															

**32-bit, acquire (size == 10 && A == 1 && R == 0)**

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, acquire and release (size == 10 && A == 1 && R == 1)**

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)**

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)**

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

**64-bit, acquire (size == 11 && A == 1 && R == 0)**

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, acquire and release (size == 11 && A == 1 && R == 1)**

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)**

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)**

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = data + value;
// All observers in the shareability domain observe the
// following load and store atomically. case op of
when
MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs				0	0	0	0	0	0	Rn				Rt						
size				V								o3				opc															

#### Acquire (A == 1 && R == 0)

```
LDADDAB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data + value;
// All observers in the shareability domain observe the
// following load and store atomically.case op of
when
MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAH and LDADDALH load from memory with acquire semantics.
- LDADDLH and LDADDALH store to memory with release semantics.
- LDADDH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0														
0		1		1		1		0		0		0		A		R		1		Rs						0		0		0		0		0		0		Rn				Rt			
size				V																o3		opc																							

#### Acquire (A == 1 && R == 0)

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data + value;
// All observers in the shareability domain observe the
// following load and store atomically.case op of
when
MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L			Rs				o0		Rt2														

### 32-bit (size == 10)

```
LDAR <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
LDAR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32; (Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, {address, dbytes, acctype}] = data;

    when
        data = MemAccType_ORDEREDMemOp_LOAD[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```



[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [ISA v82A A64 xml 00bet3.1 OPT](#)

## LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size								L			Rs				o0		Rt2														

### No offset

LDARB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, {address, dbytes, acctype}] = data;

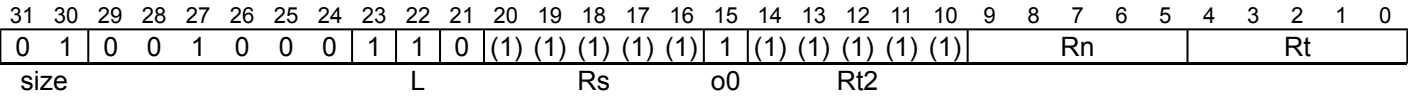
    when
        data = MemAccType_ORDEREDMemOp_LOAD; {address, dbytes, acctype};
X[t] = ZeroExtend(data, 32); {data, regsize};
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



## No offset

```
LDARH <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 2, {address, dbytes, acctype}] = data;

  when
    data = MemAccType_ORDEREDMemOp_LOAD[address, dbytes, acctype];
X[t] = ZeroExtend(data, 32); {data, regsize};
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)	1	Rt2				Rn				Rt						
								L			Rs				o0																

### 32-bit (sz == 0)

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << (Rt2); // ignored by load/store single register
integer s = UInt(sz);
integer datasize = elsize * 2; (Rs); // ignored by all loads and store-release
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDAXP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if t == t2 then if memop ==
  MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if n == 31 then if memop ==
  MemOp_STORE then
    if s == t || (pair && s == t2) then
      Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
      assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_NONE rt_unknown = FALSE; // store original value
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();
    if s == n && n != 31 then
      Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
      assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NONE rn_unknown = FALSE; // address is original base
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads. case memop of
  when
    MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
      else
        data = X[t];

  bit status = '1';
  // Check whether the Exclusive Monitors are set to include the
  // physical memory locations corresponding to virtual address
  // range [address, address+dbytes-1].
  if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, acetype] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

when MemOp_LOAD

```

```

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
elseif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, {address, dbytes, acctype}];
    if AccType ORDERED;
    if BigEndian() then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then (address, dbytes) then
        iswrite = FALSE;
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault({acctype, iswrite, secondstage}); AccType ORDERED;
        Mem[address + 0, 8, acctype];
        X[t] = [t2] = Mem[address, 8, {address + 8, 8, acctype}];
    else
        data = AccType ORDERED Mem[address, dbytes, acctype];
        X[t2] = [t] = MemZeroExtend[address+8, 8, AccType ORDERED]; (data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs				o0		Rt2														

### 32-bit (size == 10)

```
LDAXR <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
LDAXR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32; (Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release
AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.





```

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if n == 31 then if memop ==
  MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
      when Constraint_NONE rt_unknown = FALSE; // store original value
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();
  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
      when Constraint_NONE rn_unknown = FALSE; // address is original base
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads. case memop of
  when
MemOp_STORE
  if rt_unknown then
    data = bits(datasize) UNKNOWN;
  elsif pair then
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1 : el2 else el2 : el1;
  else
    data = X[t];

  bit status = '1';
  // Check whether the Exclusive Monitors are set to include the
  // physical memory locations corresponding to virtual address
  // range [address, address+dbytes-1].
  if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, acetype] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

```

```

when MemOp_LOAD
    // Tell the Exclusive Monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusive Monitor will only be set if all the reads are from the
    // same dbytes-aligned physical address, to allow for the possibility of
    // an atomicity break if the translation is changed between reads.
    AArch64.SetExclusiveMonitors(address, dbytes);

data = if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, {address, dbytes, acctype}];
        if () then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[AccType_ORDEREDBigEndian];[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-

ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L			Rs			o0			Rt2														

### No offset

```
LDAXRB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt); {Rt};
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_NONE rn_unknown = FALSE; // address is original base
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads. case memop of
    when
MemOp_STORE
    if rt_unknown then
        data = bits(datasize) UNKNOWN;
    elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
    else
        data = X[t];

    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acetype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

when MemOp_LOAD

```

```

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);
(address, dbytes);

data = if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, 1, {address, dbytes, acctype}];
        if () then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemAccType ORDEREDBigEndian[address, dbytes, acctype];
X[t] = ZeroExtend(data, 32);(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-

ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L			Rs				o0		Rt2														

### No offset

```
LDAXRH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.





```

bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_NONE rn_unknown = FALSE; // address is original base
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads. case memop of
    when
MemOp_STORE
    if rt_unknown then
        data = bits(datasize) UNKNOWN;
    elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
    else
        data = X[t];

    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acetype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

when MemOp_LOAD

```

```

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);
(address, dbytes);

data = if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, 2, {address, dbytes, acctype}];
        if () then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemAccType ORDEREDBigEndian[address, dbytes, acctype];
X[t] = ZeroExtend(data, 32);(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-

ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDCLRA and LDCLRAL load from memory with acquire semantics.
- LDCLRL and LDCLRAL store to memory with release semantics.
- LDCLR has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs					0	0	0	1	0	0	Rn					Rt				
size		V															o3		opc												

**32-bit, acquire (size == 10 && A == 1 && R == 0)**

LDCLRA <Ws>, <Wt>, [<Xn|SP>]

**32-bit, acquire and release (size == 10 && A == 1 && R == 1)**

LDCLRAL <Ws>, <Wt>, [<Xn|SP>]

**32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)**

LDCLR <Ws>, <Wt>, [<Xn|SP>]

**32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)**

LDCLRL <Ws>, <Wt>, [<Xn|SP>]

**64-bit, acquire (size == 11 && A == 1 && R == 0)**

LDCLRA <Xs>, <Xt>, [<Xn|SP>]

**64-bit, acquire and release (size == 11 && A == 1 && R == 1)**

LDCLRAL <Xs>, <Xt>, [<Xn|SP>]

**64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)**

LDCLR <Xs>, <Xt>, [<Xn|SP>]

**64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)**

LDCLRL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = data AND ease op of
when MemAtomicOp\_ADD result = data + value;
when MemAtomicOp\_BIC result = data AND NOT(value);
when MemAtomicOp\_EOR result = data EOR value;
when MemAtomicOp\_ORR result = data OR value;
when MemAtomicOp\_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp\_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp\_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp\_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp\_SWP value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDCLRB, LDCLRAB, LDCLRAB, LDCLRAB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRAB load from memory with acquire semantics.
- LDCLRAB and LDCLRAB store to memory with release semantics.
- LDCLRB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1					Rs	0	0	0	1	0	0										Rt
size				V								o3				opc								Rn				Rt			

#### Acquire (A == 1 && R == 0)

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDCLRB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data AND ease op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP (value);
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



## LDCLR<sub>H</sub>, LDCLR<sub>RAH</sub>, LDCLR<sub>RALH</sub>, LDCLR<sub>RLH</sub>

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLR<sub>RAH</sub> and LDCLR<sub>RALH</sub> load from memory with acquire semantics.
- LDCLR<sub>RLH</sub> and LDCLR<sub>RALH</sub> store to memory with release semantics.
- LDCLR<sub>RH</sub> has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
0		1		1		1		0		0		0		A		R		1		Rs						0		0		0		1		0		0		Rn						Rt			
size				V								o3												opc																							

#### Acquire (A == 1 && R == 0)

```
LDCLRRAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDCLRRALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDCLRRH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDCLRRLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data AND ease op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP (value);
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDEORA and LDEORAL load from memory with acquire semantics.
- LDEORL and LDEORAL store to memory with release semantics.
- LDEOR has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																			
1		x		1		1		1		0		0		0		A		R		1		Rs					0		0		1		0		0		0		0		Rn					Rt				
size				V												o3										opc																								

**32-bit, acquire (size == 10 && A == 1 && R == 0)**

```
LDEORA <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, acquire and release (size == 10 && A == 1 && R == 1)**

```
LDEORAL <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)**

```
LDEOR <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)**

```
LDEORL <Ws>, <Wt>, [<Xn|SP>]
```

**64-bit, acquire (size == 11 && A == 1 && R == 0)**

```
LDEORA <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, acquire and release (size == 11 && A == 1 && R == 1)**

```
LDEORAL <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)**

```
LDEOR <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)**

```
LDEORL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically. case op of
when
MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																	
0		0		1		1		1		0		0		0		A		R		1		Rs					0		0		1		0		0		0		Rn					Rt				
size				V								o3										opc																										

#### Acquire (A == 1 && R == 0)

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically.case op of
when
MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				0	0	1	0	0	0	Rn				Rt						
size					V										o3				opc												

#### Acquire (A == 1 && R == 0)

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDEORALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDEORH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDEORLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
  UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding; ();
```



## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically.case op of
when
MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

# LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

## No offset (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L			Rs					o0		Rt2													

## 32-bit (size == 10)

```
LDLAR <Wt>, [<Xn|SP>{, #0}]
```

## 64-bit (size == 11)

```
LDLAR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32; (Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
when MemOp\_STORE
data = X[t];
Mem[address, dbytes, [address, dbytes, acctype] = data;

when
data = MemAccType LIMITEDORDEREDMemOp\_LOAD; [address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in [Load LOAcquire, Store LORelease](#). For information about memory accesses, see [Load/Store addressing modes](#).

### No offset (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size								L				Rs				o0		Rt2													

### No offset

```
LDLARB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, [address, dbytes, acctype]] = data;

    when
        data = MemAccType_LIMITEDORDEREDMemOp_LOAD; [address, dbytes, acctype];
X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

### No offset (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size		L							Rs					o0		Rt2															

### No offset

```
LDLARH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, [address, dbytes, acctype]] = data;

    when
        data = MemAccType_LIMITEDORDEREDMemOp_LOAD; [address, dbytes, acctype];
X[t] = ZeroExtend(data, 32);(data, regsize);
```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	1	0	1	1	0	0	0	1	imm7							Rt2					Rn				Rt						
L																															

### 32-bit (opc == 00)

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.boolean_wback = FALSE;  
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP (SIMD&FP)*.

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.



## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then(Rt2); AccType acctype = AccType_VECSTREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then if memop ==
  MemOp_LOAD && t == t2 then
  Constraint c = ConstrainsUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;
if ! postindex then
  address = address + offset;

data1 = case memop of
  when MemOp_STORE
    data1 = V[t];
    data2 = V[t2];
    Mem[address, dbytes, {address + 0, dbytes, acctype}] = data1; AccType_VECSTREAMMem];
data2 = {address + dbytes, dbytes, acctype} = data2;

  when MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, {address + 0, dbytes, acctype}];
    data2 = AccType_VECSTREAMMem];
if rt_unknown then
  data1 = bits(datasize) UNKNOWN;
  data2 = bits(datasize) UNKNOWN; {address + dbytes, dbytes, acctype};
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;

if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[t2] = data2; {n} = address;
```

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

## LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see [Load/Store addressing modes](#). For information about Non-temporal pair instructions, see [Load/Store Non-temporal pair](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	0	1	0	1	0	0	0	0	1	imm7							Rt2				Rn				Rt						
opc										L																					

### 32-bit (opc == 00)

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 10)

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDNP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then(Rt2); AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then if memop ==
  MemOp_LOAD && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;
if ! postindex then
  address = address + offset;

data1 = case memop of
  when MemOp_STORE
    if rt_unknown && t == n then
      data1 = bits(datasize) UNKNOWN;
    else
      data1 = X[t];
    if rt_unknown && t2 == n then
      data2 = bits(datasize) UNKNOWN;
    else
      data2 = X[t2];
  Mem[address, dbytes, {address + 0, dbytes, acctype}] = data1; AccType_STREAMMem];
data2 = {address + dbytes, dbytes, acctype} = data2;

  when MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, {address + 0, dbytes, acctype}];
    data2 = AccType_STREAMMem];
if rt_unknown then
  data1 = bits(datasize) UNKNOWN;
  data2 = bits(datasize) UNKNOWN; {address + dbytes, dbytes, acctype};
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
X[t] = data1;
X[t2] = data2;

if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[t] = data1; {} = address;
  else
    X[t2] = data2; {n} = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-

ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
opc		1		0		1		1		0		0		1		1		imm7							Rt2					Rn					Rt			
L																																						

#### 32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>], #<imm>
```

#### 64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

#### 128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;  
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
opc		1		0		1		1		0		1		1		1		imm7							Rt2					Rn					Rt			
L																																						

#### 32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

#### 64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

#### 128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;  
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	0	1	imm7							Rt2					Rn					Rt				

### 32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDP \(SIMD&FP\)](#).

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	<p>For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as &lt;imm&gt;/16.</p> <p>For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/16.</p>

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
if opc == '11' then (Rt2); AccType acctype = AccType_VEC;  
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
if opc == '11' then UnallocatedEncoding();  
integer scale = 2 + UInt(opc);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then
    if memop == MemOp_LOAD && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    if !postindex then
        address = address + offset;

data1 = case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address, dbytes, {address + 0, dbytes, acctype}] = data1; AccType_VECMem];
data2 = {address + dbytes, dbytes, acctype} = data2;

    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes, {address + 0, dbytes, acctype}];
        data2 = AccType_VECMem];

if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN; {address + dbytes, dbytes, acctype};
    if rt_unknown then
        data1 = bits(datasize) UNKNOWN;
        data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	0	1	0	1	0	0	0	1	1	imm7							Rt2				Rn				Rt						
opc										L																					

#### 32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

#### 64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
x		0	1		0	1		0	0		1	1		1		imm7							Rt2				Rn				Rt			
opc										L																								

#### 32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

#### 64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
x		0		1		0		1		0		0		1		imm7							Rt2				Rn				Rt			
opc										L																								



### 32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	<p>For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/8.</p>

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
if L:opc<0> == '01' || opc == '11' then (Rt2); AccType acctype = AccType_NORMAL;  
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();  
boolean signed = (opc<0> != '0');  
integer scale = 2 + UInt(opc<1>);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```



```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
  if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
      when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if t == t2 then
    if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
      Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
      assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

    if memop == MemOp_LOAD && t == t2 then
      Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
      assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  if !postindex then
    address = address + offset;

data1 = case memop of
  when MemOp_STORE
    if rt_unknown && t == n then
      data1 = bits(datasize) UNKNOWN;
    else
      data1 = X[t];
    if rt_unknown && t2 == n then
      data2 = bits(datasize) UNKNOWN;
    else
      data2 = X[t2];
  Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_NORMALMem];
data2 = [address + dbytes, dbytes, acctype] = data2;

  when MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
    data2 = AccType_NORMALMem];
if rt_unknown then
  data1 = bits(datasize) UNKNOWN;
  data2 = bits(datasize) UNKNOWN;
if signed then [address + dbytes, dbytes, acctype];
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
  if signed then
    X[t] = SignExtend(data1, 64);

```

```

    X\[t2\] = SignExtend(data2, 64);
else
    X\[t\] = data1;
    X\[t2\] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	0	1	1	imm7							Rt2				Rn				Rt						
opc										L																					

### Post-index

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	1	1	imm7							Rt2				Rn				Rt						
opc										L																					

### Pre-index

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	0	1	imm7							Rt2				Rn				Rt						
opc										L																					

### Signed offset

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDPSW](#).

## Assembler Symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bits(64) offset = (Rt2); AccType acctype = AccType NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), 2); {imm7, -64}, scale);
```



```

bits(64) address;
bits(32) data1;
bits(32) data2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then if memop ==
  MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
  assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint UNDEF UnallocatedEncoding();
    when Constraint NOP EndOfInstruction();

if t == t2 then if memop ==
  MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
  assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint NONE rt_unknown = FALSE; // value stored is pre-writeback
    when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint UNDEF UnallocatedEncoding();
    when Constraint NOP EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint UNDEF UnallocatedEncoding();
    when Constraint NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
if ! postindex then
  address = address + offset;

data1 = case memop of
  when MemOp_STORE
    if rt_unknown && t == n then
      data1 = bits(datasize) UNKNOWN;
    else
      data1 = X[t];
    if rt_unknown && t2 == n then
      data2 = bits(datasize) UNKNOWN;
    else
      data2 = X[t2];
  Mem[address, 4, [address + 0, dbytes, acctype] = data1; AccType NORMALMem];
data2 = [address + dbytes, dbytes, acctype] = data2;

  when MemOp_LOAD
    data1 = Mem[address+4, 4, [address + 0, dbytes, acctype];
    data2 = AccType NORMALMem];
if rt_unknown then
  data1 = bits(32) UNKNOWN;
  data2 = bits(32) UNKNOWN; [address + dbytes, dbytes, acctype];
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
  if signed then

```



```

X[t] = SignExtend(data1, 64);
X[t2] = SignExtend(data2, 64);
else
    X[t] = data1;
    X(data2, 64);
[t2] = data2;
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	1	0	imm9									0	1	Rn				Rt					
opc																															

#### 8-bit (size == 00 && opc == 01)

LDR <Bt>, [<Xn|SP>], #<sim>

#### 16-bit (size == 01 && opc == 01)

LDR <Ht>, [<Xn|SP>], #<sim>

#### 32-bit (size == 10 && opc == 01)

LDR <St>, [<Xn|SP>], #<sim>

#### 64-bit (size == 11 && opc == 01)

LDR <Dt>, [<Xn|SP>], #<sim>

#### 128-bit (size == 00 && opc == 11)

LDR <Qt>, [<Xn|SP>], #<sim>

```
boolean wback = TRUE;  
boolean postindex = TRUE;  
integer scale = UInt(opc<1>:size);  
if scale > 4 then UnallocatedEncoding();  
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	1	0	imm9									1	1	Rn				Rt					
opc																															

8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>, #<sim>]!
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, #<sim>]!
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, #<sim>]!
```

64-bit (size == 11 && opc == 01)

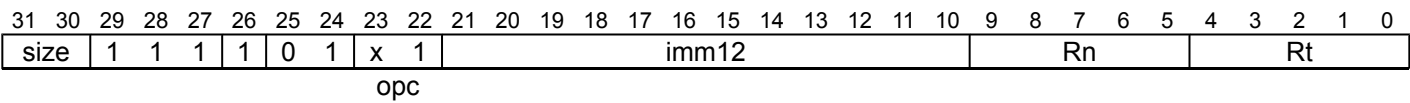
```
LDR <Dt>, [<Xn|SP>, #<sim>]!
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	<p>For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.</p> <p>For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/2.</p> <p>For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/4.</p> <p>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/8.</p> <p>For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/16.</p>

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

[ISA v82A A64 xml 00bet3.1](#) [htmldiff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#). The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	1	0	imm9									0	1	Rn				Rt									
size										opc																									

#### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;  
boolean postindex = TRUE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn				Rt									
size										opc																									

#### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
1		x		1		1		1		0		0		1		0		1		imm12											Rn				Rt			
size										opc																												

### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDR \(immediate\)](#).

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.  For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer regsize;  
  
regsize = if size == '11' then 64 else 32;  
integer datasize = 8 << scale; (Rt); AccType acctype = AccType_NORMAL;  
MemOp memop;  
boolean signed;  
integer regsize;  
  
if opc<1> == '0' then  
    // store or zero-extending load  
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;  
    regsize = if size == '11' then 64 else 32;  
    signed = FALSE;  
else  
    if size == '11' then  
        UnallocatedEncoding();  
    else  
        // sign-extending load  
        memop = MemOp_LOAD;  
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();  
        regsize = if opc<0> == '1' then 32 else 64;  
        signed = TRUE;  
  
integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then if memop ==
    MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype} = data;

    when AccType_NORMAL MemOp_LOAD; data =
        Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(data, regsize);
    {address, t<4:0>};

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```



[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
size		1		1		1		1		0		0		x		1		1		Rm						option		S		1		0		Rn						Rt	
opc																																									

### 8-bit (size == 00 && opc == 01 && option != 011)

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 8-bit (size == 00 && opc == 01 && option == 011)

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### 16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SCTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt (Rn) ;
integer t = UInt (Rt) ;
integer m = UInt (Rm) ;
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if !postindex then
    address = address + offset;

case memop of
    when MemOp\_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp\_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	1	1	Rm				option			S	1	0	Rn				Rt										
size										opc																									

### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
if option<1> == '0' then UnallocatedEncoding(); // sub-word index  
ExtendType extend_type = DecodeRegExtend(option);  
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount>	For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":
----------	---

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale; (Rm); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if ope<1> == '0' then
    // store or zero-extending load
    memop = if ope<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if ope<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && ope<0> == '1' then UnallocatedEncoding();
        regsize = if ope<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, regsize); [n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.





## LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9									0	1	Rn				Rt					
size											opc																				

### Post-index

```
LDRB <Wt>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn				Rt					
size											opc																				

### Pre-index

```
LDRB <Wt>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
0		0		1		1		1		0		0		1		0		1		imm12											Rn					Rt				
size											opc																													

### Unsigned offset

```
LDRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0); (imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRH \(immediate\)](#).

## Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if ope<1> == '0' then
    // store or zero-extending load
    memop = if ope<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && ope<0> == '1' then UnallocatedEncoding();
        regsize = if ope<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;

```

## Operation

```

bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if n == 31 then if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE; // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

    when AccType NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(data, 32);
(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
0	0	1	1	1	0	0	0	0	1	1	Rm				option			S	1	0	Rn				Rt											
size											opc																									

### Extended register (option != 011)

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### Shifted register (option == 011)

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
if option<1> == '0' then UnallocatedEncoding(); // sub-word index  
ExtendType extend_type = DecodeRegExtend(option);  
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

<amount>	Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.
----------	---

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm); Acctype acctype = Acctype_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    address = address + offset;
    if ! postindex then
        address = address + offset;

    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
            Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

        when AccType_NORMAL MemOp_LOAD; data =
            Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X(data, 32); [n] = address;

```

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>



## LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	1	0	imm9									0	1	Rn				Rt									
size										opc																									

### Post-index

```
LDRH <Wt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = integer-scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn				Rt									
size										opc																									

### Pre-index

```
LDRH <Wt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = integer-scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
0		1		1		1		0		0		1		0		1		imm12												Rn				Rt			
size										opc																											

### Unsigned offset

```
LDRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer-scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1); (imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRH \(immediate\)](#).

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if n == 31 then if memop ==
    MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE; // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

    when AccType NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(data, 32);
(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	1	1	Rm				option			S	1	0	Rn				Rt						
size											opc																				

### 32-bit

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0; integer shift = if S == '1' then scale else 0;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>	Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":
----------	---

S	<amount>
0	#0
1	#1

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm); Acctype acctype = Acctype_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32); [n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA_v82A_A64_xml_00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA_v82A_A64_xml_00bet3.1 OPT</u></a>



## LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	1	x	0	imm9									0	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;  
boolean postindex = TRUE;  
bits(64) offset = integer-scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	1	x	0	imm9									1	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;  
boolean postindex = FALSE;  
bits(64) offset = integer-scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
0		0		1		1		1		0		0		1		1		x		imm12											Rn				Rt			
size										opc																												

### 32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer-scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0); (imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSB \(immediate\)](#).

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE;      // writeback is suppressed
    when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
    when Constraint_UNDEF      UnallocatedEncoding();
    when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE      rt_unknown = FALSE;   // value stored is original value
    when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
    when Constraint_UNDEF      UnallocatedEncoding();
    when Constraint_NOP        EndOfInstruction();

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    if rt_unknown then
      data = bits(8) UNKNOWN;
data = bits(datasize) UNKNOWN;
    else
      data = X[t];
      Mem[address, 1, [address, datasize DIV 8, acctype] = data;
when AccType_NORMAL] = data;
  when MemOp_LOAD
    data = Mem[address, 1, AccType_NORMAL];
[address, datasize DIV 8, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

```

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

## LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	1	x	1	Rm					option			S	1	0	Rn					Rt				
size								opc																							

### 32-bit with extended register offset (opc == 11 && option != 011)

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 32-bit with shifted register offset (opc == 11 && option == 011)

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### 64-bit with extended register offset (opc == 10 && option != 011)

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 64-bit with shifted register offset (opc == 10 && option == 011)

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

<amount>	Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.
----------	---

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop != if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstraintUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstraintUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    address = address + offset;
    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp_STORE
            data = if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
            Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

        when AccType_NORMAL] = data;

        when MemOp_LOAD
            data = Mem[address, 1, {address, datasize DIV 8, acctype}];
            if signed then AccType_NORMAL];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCHPrefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X(address, t<4:0>)[n] = address;

```

[ISA v82A A64 xml 00bet3.1](#) [htmldiff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)



## LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	1	x	0	imm9									0	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;  
boolean postindex = TRUE;  
bits(64) offset = integer-scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	1	x	0	imm9									1	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;  
boolean postindex = FALSE;  
bits(64) offset = integer-scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0		1		1		1		1		0		0		1		1		x		imm12												Rn				Rt			
size										opc																													

### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer-scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1); (imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSH \(immediate\)](#).

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then UnallocatedEncoding();
    else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
    when Constraint_UNKNOWN    wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF      UnallocatedEncoding();
    when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE    rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE;  // value stored is UNKNOWN
    when Constraint_UNDEF    UnallocatedEncoding();
    when Constraint_NOP      EndOfInstruction();

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    if rt_unknown then
      data = bits(16) UNKNOWN;
data = bits(datasize) UNKNOWN;
    else
      data = X[t];
      Mem[address, 2, [address, datasize DIV 8, acctype] = data;
when AccType_NORMAL] = data;
  when MemOp_LOAD
    data = Mem[address, 2, AccType_NORMAL];
[address, datasize DIV 8, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
```

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

## LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	1	Rm				option			S	1	0	Rn				Rt						
size								opc																							

### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0; integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount>	Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":
----------	---

S	<amount>
0	#0
1	#1

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop != MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    address = address + offset;
    if ! postindex then
        address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, {address, datasize DIV 8, acctype}];
        if signed then AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(address, t<4:0>);[n] = address;

```

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>



## LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	1	1	0	0	0	1	0	0	imm9									0	1	Rn				Rt									
size										opc																									

### Post-index

```
LDRSW <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	1	1	0	0	0	1	0	0	imm9									1	1	Rn				Rt									
size										opc																									

### Pre-index

```
LDRSW <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
1		0		1		1		1		0		0		1		1		0		imm12												Rn						Rt			
size										opc																															

### Unsigned offset

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2); (imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSW \(immediate\)](#).

## Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simmm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(32) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if n == 31 then if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE; // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 4, {address, datasize DIV 8, acctype}] = data;

    when AccType NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(data, 64);
(address, t<4:0>);

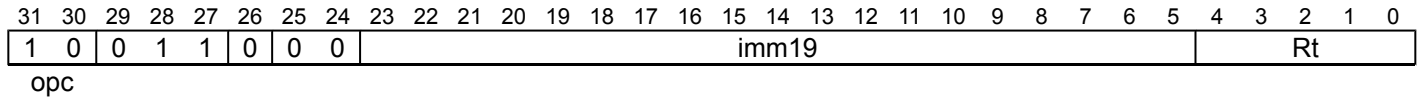
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

<a href="#"><u>ISA_v82A_A64_xml_00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA_v82A_A64_xml_00bet3.1 OPT</u></a>

## LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



### Literal

LDRSW <Xt>, <label>

```
integer t = UInt(Rt);
bits(64) offset;

offset = (Rt); MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

### Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

### Operation

```
bits(64) address = PC[] + offset;
bits(32) data;
bits(size*8) data;

data = case memop of
  when MemOp_LOAD
    data = Mem[address, 4, {address, size, AccType_NORMAL}];
    if signed then
      X[t] = SignExtend(data, 64);
    else
      X[t] = data;

  when MemOp_PREFETCH
    Prefetch(data, 64); {address, t<4:0>};
```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	0	1	1	1	0	0	0	1	0	1	Rm					option			S	1	0	Rn					Rt									
size											opc																									

### 64-bit

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0; integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>	Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":
----------	---

S	<amount>
0	#0
1	#2

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm); Acctype acctype = Acctype_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```



## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(32) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 4, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 64)[n] = address;

```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- LDSET has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer  
(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	0	1	1	0	0	Rn				Rt						
size		V								o3				opc																	

**32-bit, acquire (size == 10 && A == 1 && R == 0)**

```
LDSETA <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, acquire and release (size == 10 && A == 1 && R == 1)**

```
LDSETAL <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)**

```
LDSET <Ws>, <Wt>, [<Xn|SP>]
```

**32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)**

```
LDSETL <Ws>, <Wt>, [<Xn|SP>]
```

**64-bit, acquire (size == 11 && A == 1 && R == 0)**

```
LDSETA <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, acquire and release (size == 11 && A == 1 && R == 1)**

```
LDSETAL <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)**

```
LDSET <Xs>, <Xt>, [<Xn|SP>]
```

**64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)**

```
LDSETL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically. case op of
when
MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	0	1	1	0	0	Rn					Rt				
size				V														o3		opc											

#### Acquire (A == 1 && R == 0)

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically. case op of
    when
MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX    result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN    result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX    result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN    result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- LDSETH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1					Rs	0	0	1	1	0	0										Rt
size				V								o3				opc								Rn				Rt			

#### Acquire (A == 1 && R == 0)

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```



## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically. case op of
    when
MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAXL and LDSMAXAL store to memory with release semantics.
- LDSMAX has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer  
(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	1	0	0	0	0	Rn				Rt						
size		V										o3				opc															

**32-bit, acquire (size == 10 && A == 1 && R == 0)**

LDSMAXA <Ws>, <Wt>, [<Xn|SP>]

**32-bit, acquire and release (size == 10 && A == 1 && R == 1)**

LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]

**32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)**

LDSMAX <Ws>, <Wt>, [<Xn|SP>]

**32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)**

LDSMAXL <Ws>, <Wt>, [<Xn|SP>]

**64-bit, acquire (size == 11 && A == 1 && R == 0)**

LDSMAXA <Xs>, <Xt>, [<Xn|SP>]

**64-bit, acquire and release (size == 11 && A == 1 && R == 1)**

LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]

**64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)**

LDSMAX <Xs>, <Xt>, [<Xn|SP>]

**64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)**

LDSMAXL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	0	0	0	0	Rn					Rt				
size					V										o3					opc											

#### Acquire (A == 1 && R == 0)

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	0	0	0	0	Rn					Rt				
size					V										o3					opc											

#### Acquire (A == 1 && R == 0)

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



## LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMINA and LDSMINAL load from memory with acquire semantics.
- LDSMINL and LDSMINAL store to memory with release semantics.
- LDSMIN has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0														
1		x		1		1		1		0		0		0		A		R		1		Rs				0		1		0		1		0		0		Rn				Rt			
size				V																o3				opc																					

**32-bit, acquire (size == 10 && A == 1 && R == 0)**

LDSMINA <Ws>, <Wt>, [<Xn|SP>]

**32-bit, acquire and release (size == 10 && A == 1 && R == 1)**

LDSMINAL <Ws>, <Wt>, [<Xn|SP>]

**32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)**

LDSMIN <Ws>, <Wt>, [<Xn|SP>]

**32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)**

LDSMINL <Ws>, <Wt>, [<Xn|SP>]

**64-bit, acquire (size == 11 && A == 1 && R == 0)**

LDSMINA <Xs>, <Xt>, [<Xn|SP>]

**64-bit, acquire and release (size == 11 && A == 1 && R == 1)**

LDSMINAL <Xs>, <Xt>, [<Xn|SP>]

**64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)**

LDSMIN <Xs>, <Xt>, [<Xn|SP>]

**64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)**

LDSMINL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	0	1	0	0	Rn					Rt				
size		V															o3			opc											

#### Acquire (A == 1 && R == 0)

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP (value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	0	1	0	0	Rn					Rt				
size					V										o3					opc											

#### Acquire (A == 1 && R == 0)

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

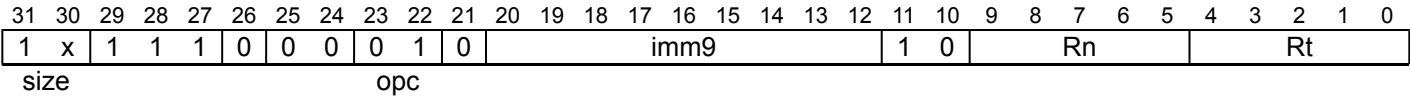
LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit (size == 10)

```
LDTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
LDTR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>
- Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>
- Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>
- Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm>
- Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.



## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale; (Rt); AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

    when AccType_UNPRIVMemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, regsize); [n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

ISA v82A A64 xml 00bet3.1  
(old)

htmldiff from- (new)  
ISA\_v82A\_A64\_xml\_00bet3.1 ISA v82A A64 xml 00bet3.1 OPT

## LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9									1	0	Rn				Rt					
size											opc																				

### Unscaled offset

```
LDTRB <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, 1, [address, datasize DIV 8, acctype]] = data;

    when AccType_UNPRIVMemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32); [n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	1	0	imm9									1	0	Rn				Rt									
size										opc																									

### Unscaled offset

```
LDTRH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, 2, [address, datasize DIV 8, acctype]] = data;

    when AccType_UNPRIVMemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32); [n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)



## LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	1	x	0	imm9									1	0	Rn				Rt									
size										opc																									

### 32-bit (opc == 11)

```
LDTRSB <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (opc == 10)

```
LDTRSB <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then UnallocatedEncoding();
    else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop != if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, [address, datasize DIV 8, acctype]] = data;

        when AccType_UNPRIV] = data;

    when MemOp_LOAD
        data = Mem[address, 1, [address, datasize DIV 8, acctype]];
        if signed then AccType_UNPRIV];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(address, t<4:0>); [n] = address;

```

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from-

[\(new\)](#)

ISA\_v82A\_A64\_xml\_00bet3.1

[ISA v82A A64 xml 00bet3.1 OPT](#)

## LDTRSH

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	0	imm9									1	0	Rn				Rt					
size										opc																					

### 32-bit (opc == 11)

```
LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]
```

### 64-bit (opc == 10)

```
LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
  if memop != if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
      when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_NONE rt_unknown = FALSE; // value stored is original value
      when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
  else
    address = X[n];

address = address + offset;
if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    data = if rt_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
    Mem[address, 2, [address, datasize DIV 8, acctype]] = data;

  when AccType_UNPRIV] = data;

  when MemOp_LOAD
    data = Mem[address, 2, [address, datasize DIV 8, acctype]];
    if signed then AccType_UNPRIV];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X(address, t<4:0>); [n] = address;

```

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)



## LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see

[Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	0	1	0	0	imm9									1	0	Rn				Rt					
size											opc																				

### Unscaled offset

```
LDTRSW <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(32) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, 4, [address, datasize DIV 8, acctype] = data;

    when AccType_UNPRIVMemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 64); [n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMAXA and LDUMAXAL load from memory with acquire semantics.
- LDUMAXL and LDUMAXAL store to memory with release semantics.
- LDUMAX has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

Integer  
(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	1	1	0	0	0	Rn				Rt						
size				V								o3				opc															

**32-bit, acquire (size == 10 && A == 1 && R == 0)**

LDUMAXA <Ws>, <Wt>, [<Xn|SP>]

**32-bit, acquire and release (size == 10 && A == 1 && R == 1)**

LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]

**32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)**

LDUMAX <Ws>, <Wt>, [<Xn|SP>]

**32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)**

LDUMAXL <Ws>, <Wt>, [<Xn|SP>]

**64-bit, acquire (size == 11 && A == 1 && R == 0)**

LDUMAXA <Xs>, <Xt>, [<Xn|SP>]

**64-bit, acquire and release (size == 11 && A == 1 && R == 1)**

LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]

**64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)**

LDUMAX <Xs>, <Xt>, [<Xn|SP>]

**64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)**

LDUMAXL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDUMAXB, LDUMAXB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	1	0	0	0	Rn					Rt				
size					V										o3					opc											

#### Acquire (A == 1 && R == 0)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



## LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	1	0	0	0	Rn					Rt				
size					V										o3					opc											

#### Acquire (A == 1 && R == 0)

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMINA and LDUMINAL load from memory with acquire semantics.
- LDUMINL and LDUMINAL store to memory with release semantics.
- LDUMIN has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

Integer  
(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	1	1	1	0	0	Rn				Rt						
size		V										o3				opc															

**32-bit, acquire (size == 10 && A == 1 && R == 0)**

LDUMINA <Ws>, <Wt>, [<Xn|SP>]

**32-bit, acquire and release (size == 10 && A == 1 && R == 1)**

LDUMINAL <Ws>, <Wt>, [<Xn|SP>]

**32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)**

LDUMIN <Ws>, <Wt>, [<Xn|SP>]

**32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)**

LDUMINL <Ws>, <Wt>, [<Xn|SP>]

**64-bit, acquire (size == 11 && A == 1 && R == 0)**

LDUMINA <Xs>, <Xt>, [<Xn|SP>]

**64-bit, acquire and release (size == 11 && A == 1 && R == 1)**

LDUMINAL <Xs>, <Xt>, [<Xn|SP>]

**64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)**

LDUMIN <Xs>, <Xt>, [<Xn|SP>]

**64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)**

LDUMINL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
- LDUMINLB and LDUMINALB store to memory with release semantics.
- LDUMINB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	1	1	0	0	Rn					Rt				
size				V								o3				opc															

#### Acquire (A == 1 && R == 0)

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	1	1	0	0	Rn					Rt				
size					V										o3					opc											

#### Acquire (A == 1 && R == 0)

```
LDUMINAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
LDUMINALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDUMINH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1 && Rt != 11111)

```
LDUMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```



## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

result = ifcase op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	1	0	imm9										0	0	Rn				Rt				
opc																															

### 8-bit (size == 00 && opc == 01)

```
LDUR <Bt>, [<Xn|SP>{, #<sim>}]
```

### 16-bit (size == 01 && opc == 01)

```
LDUR <Ht>, [<Xn|SP>{, #<sim>}]
```

### 32-bit (size == 10 && opc == 01)

```
LDUR <St>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (size == 11 && opc == 01)

```
LDUR <Dt>, [<Xn|SP>{, #<sim>}]
```

### 128-bit (size == 00 && opc == 11)

```
LDUR <Qt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(opc<1>:size);  
if scale > 4 then UnallocatedEncoding();  
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
AccType acctype = AccType_VEC;  
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;  
integer datasize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64() ;

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

case memop of
    when MemOp\_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp\_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	1	0	imm9									0	0	Rn				Rt									
size										opc																									

### 32-bit (size == 10)

```
LDUR <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (size == 11)

```
LDUR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale; (Rt); AccType acctype = AccType-NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
  MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
      when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;
if ! postindex then
  address = address + offset;

data = case memop of
  when MemOp_STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
  Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

  when AccType NORMAL MemOp_LOAD; data =
  Mem[address, datasize DIV 8, acctype];
  if signed then
    X[t] = SignExtend(data, regsize);
  else
    X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X(data, regsize); [n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

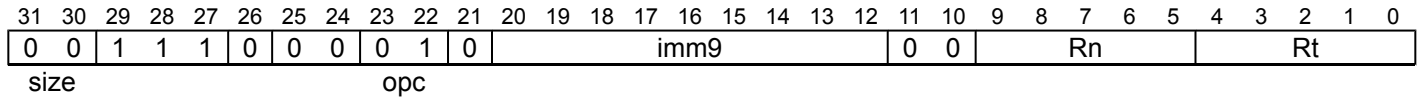
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

ISA v82A A64 xml 00bet3.1  
(old)

htmldiff from- (new)  
ISA\_v82A\_A64\_xml\_00bet3.1 ISA v82A A64 xml 00bet3.1 OPT

## LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



### Unscaled offset

```
LDURB <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, 1, [address, datasize DIV 8, acctype]] = data;

    when AccType_NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32); [n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)



## LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	1	0	imm9										0	0	Rn				Rt								
size										opc																									

### Unscaled offset

```
LDURH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, 2, [address, datasize DIV 8, acctype]] = data;

    when AccType_NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32); [n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
0		0		1		1		1		0		0		0		1		x		0		imm9									0		0		Rn				Rt			
size										opc																																

### 32-bit (opc == 11)

```
LDURSB <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (opc == 10)

```
LDURSB <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset =boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
  if memop != if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
      when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_NONE rt_unknown = FALSE; // value stored is original value
      when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
  else
    address = X[n];

address = address + offset;
if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    data = if rt_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
    Mem[address, 1, [address, datasize DIV 8, acctype] = data;

    when AccType_NORMAL] = data;

  when MemOp_LOAD
    data = Mem[address, 1, [address, datasize DIV 8, acctype];
    if signed then AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X(address, t<4:0>); [n] = address;

```

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from-

[\(new\)](#)

ISA\_v82A\_A64\_xml\_00bet3.1

[ISA v82A A64 xml 00bet3.1 OPT](#)

## LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	1	x	0	imm9									0	0	Rn				Rt									
size										opc																									

### 32-bit (opc == 11)

```
LDURSH <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (opc == 10)

```
LDURSH <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset =boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.



## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
  if memop != if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
      when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_NONE rt_unknown = FALSE; // value stored is original value
      when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
  else
    address = X[n];

address = address + offset;
if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    data = if rt_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
    Mem[address, 2, [address, datasize DIV 8, acctype] = data;

    when AccType_NORMAL] = data;

  when MemOp_LOAD
    data = Mem[address, 2, [address, datasize DIV 8, acctype];
    if signed then AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X(address, t<4:0>); [n] = address;

```

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from-

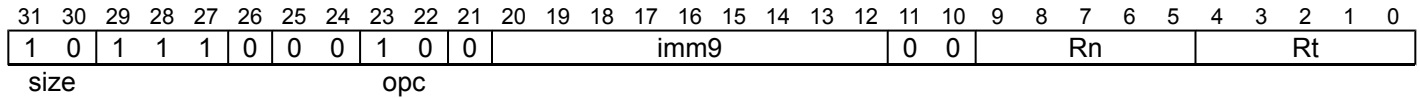
[\(new\)](#)

ISA\_v82A\_A64\_xml\_00bet3.1

[ISA v82A A64 xml 00bet3.1 OPT](#)

## LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



### Unscaled offset

```
LDURSW <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(32) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, 4, [address, datasize DIV 8, acctype]] = data;

    when AccType_NORMAL MemOp_LOAD; data =
Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 64); [n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)	0	Rt2				Rn				Rt						
								L			Rs				o0																

### 32-bit (sz == 0)

```
LDXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
LDXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << (Rt2); // ignored by load/store single register
integer s = UInt(sz);
integer datasize = elsize * 2; (Rs); // ignored by all loads and store-release
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDXP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.





```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if t == t2 then if memop ==
  MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if n == 31 then if memop ==
  MemOp_STORE then
    if s == t || (pair && s == t2) then
      Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
      assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_NONE rt_unknown = FALSE; // store original value
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();
    if s == n && n != 31 then
      Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
      assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NONE rn_unknown = FALSE; // address is original base
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads. case memop of
  when
    MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
      else
        data = X[t];

  bit status = '1';
  // Check whether the Exclusive Monitors are set to include the
  // physical memory locations corresponding to virtual address
  // range [address, address+dbytes-1].
  if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, acetype] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

when MemOp_LOAD

```

```

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
elseif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, {address, dbytes, acctype}];
    if AccType ATOMIC;
    if BigEndian() then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then (address, dbytes) then
        iswrite = FALSE;
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault({acctype, iswrite, secondstage}); AccType ATOMIC;
        Mem[address + 0, 8, acctype];
        X[t] = [t2] = Mem[address, 8, {address + 8, 8, acctype}];
    else
        data = AccType ATOMICMem[address, dbytes, acctype];
        X[t2] = [t] = MemZeroExtend[address+8, 8, AccType ATOMIC]; (data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L			Rs				o0		Rt2														

### 32-bit (size == 10)

```
LDXR <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
LDXR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32; (Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
            when Constraint_UNDEF       UnallocatedEncoding();
            when Constraint_NOP         EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;    // store original value
            when Constraint_UNDEF       UnallocatedEncoding();
            when Constraint_NOP         EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;    // address is original base
            when Constraint_UNDEF       UnallocatedEncoding();
            when Constraint_NOP         EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads. case memop of
    when
MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acetype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

```

```

when MemOp_LOAD
    // Tell the Exclusive Monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusive Monitor will only be set if all the reads are from the
    // same dbytes-aligned physical address, to allow for the possibility of
    // an atomicity break if the translation is changed between reads.
    AArch64.SetExclusiveMonitors(address, dbytes);

data = if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, {address, dbytes, acctype}];
        if () then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemAccType_ATOMICBigEndian[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L			Rs				o0		Rt2														

### No offset

```
LDXRB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.





```

bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_NONE rn_unknown = FALSE; // address is original base
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads. case memop of
    when
MemOp_STORE
    if rt_unknown then
        data = bits(datasize) UNKNOWN;
    elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
    else
        data = X[t];

    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acetype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

when MemOp_LOAD

```

```

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);
(address, dbytes);

data = if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, 1, {address, dbytes, acctype}];
        if () then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemAccType ATOMICBigEndian[address, dbytes, acctype];
X[t] = ZeroExtend(data, 32);(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-

ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs				o0		Rt2														

### No offset

```
LDXRH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_NONE rn_unknown = FALSE; // address is original base
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads. case memop of
    when
MemOp_STORE
    if rt_unknown then
        data = bits(datasize) UNKNOWN;
    elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
    else
        data = X[t];

    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acetype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

when MemOp_LOAD

```

```

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);
(address, dbytes);

data = if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, 2, {address, dbytes, acctype}];
        if () then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemAccType ATOMICBigEndian[address, dbytes, acctype];
X[t] = ZeroExtend(data, 32);(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-

ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias [MUL](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	1	0	0	0	Rm				0	Ra				Rn				Rd							
o0																															

### 32-bit (sf == 0)

MADD <Wd>, <Wn>, <Wm>, <Wa>

### 64-bit (sf == 1)

MADD <Xd>, <Xn>, <Xm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32; integer destsize = if sf == '1' then 64 else 32;
integer datasize = destsize;
boolean sub_op = (o0 == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">MUL</a>	Ra == '11111'

## Operation

```
bits(destsize) operand1 =bits(datasize) operand1 = X[n];  
bits(destsize) operand2 =bits(datasize) operand2 = X[m];  
bits(destsize) operand3 = X[a];  
  
integer result;  
  
result =if sub_op then  
result = UInt(operand3) + ((operand3) - (UInt(operand1) * UInt(operand2))); (operand2);  
else  
result =  
  
UInt(operand3) + (UInt(operand1) * UInt(operand2));  
  
X[d] = result<destsize-1:0>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

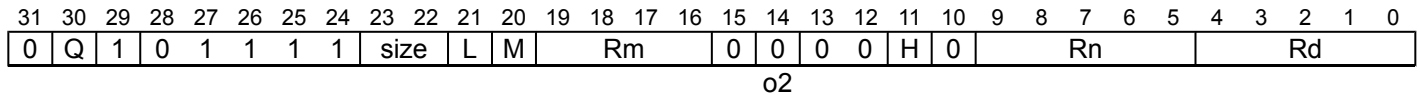
<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



## MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Vector

MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
product = (element1 * element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

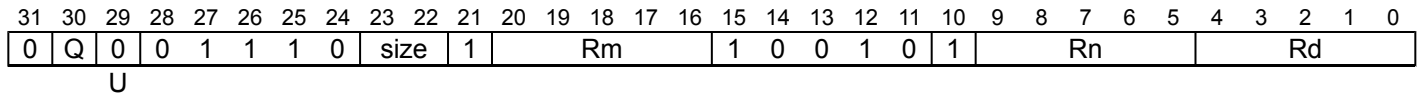
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

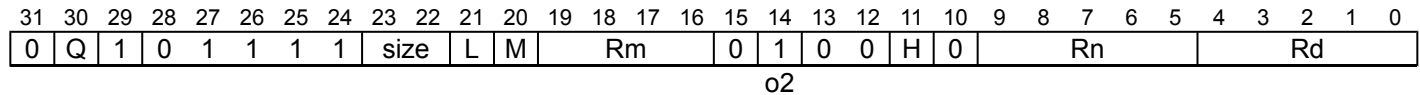
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Vector

MLS [<Vd>.<T>](#), [<Vn>.<T>](#), [<Vm>.<Ts>](#)[[<index>](#)]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

## Assembler Symbols

[<Vd>](#) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

[<T>](#) Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

[<Vn>](#) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

[<Vm>](#) Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size [<Ts>](#) is H.

[<Ts>](#) Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
product = (element1 * element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

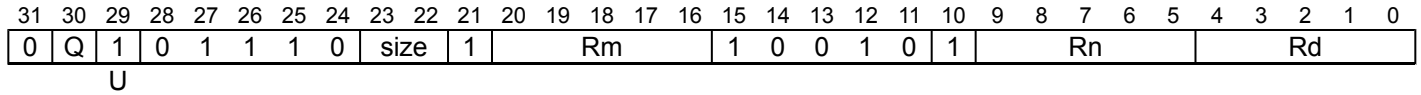
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## MLS (vector)

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



## MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		1		1		1		0		0		1		0		1		hw		imm16																		Rd			
opc																																									

### 32-bit (sf == 0)

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

### 64-bit (sf == 1)

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then bits(16) imm = imm16;
integer pos; MoveWideOp opcode;

case opc of
  when '00' opcode = MoveWideOp_N;
  when '10' opcode = MoveWideOp_Z;
  when '11' opcode = MoveWideOp_K;
  otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Operation

```
bits(datasize) result;

result = if opcode == MoveWideOp_K then
  result = X[d];
else
  result = Zeros();

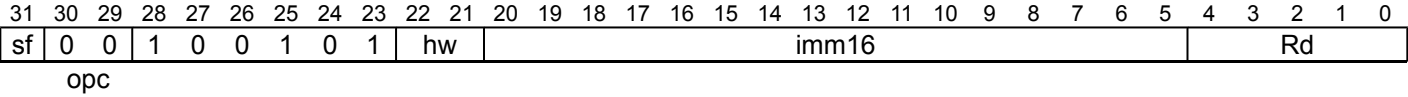
result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
  result = NOT[d];
result<pos+15:pos> = imm16; {result};
X[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#) [htmldiff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

# MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias [MOV \(inverted wide immediate\)](#).



## 32-bit (sf == 0)

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

## 64-bit (sf == 1)

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then bits(16) imm = imm16;
integer pos; MoveWideOp opcode;

case opcode of
  when '00' opcode = MoveWideOp_N;
  when '10' opcode = MoveWideOp_Z;
  when '11' opcode = MoveWideOp_K;
  otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Alias Conditions

Alias	Of variant Is preferred when	
<a href="#">MOV (inverted wide immediate)</a>	64-bit	! (IsZero(imm16) && hw != '00')
<a href="#">MOV (inverted wide immediate)</a>	32-bit	! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)

## Operation

```
bits(datasize) result;

result =if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N();

result<pos+15:pos> = imm16;
result =then
    result = NOT(result);
X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

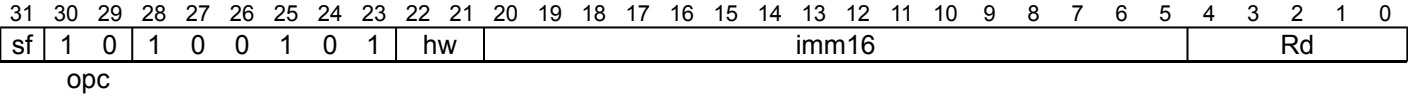
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

# MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias [MOV \(wide immediate\)](#).



## 32-bit (sf == 0)

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

## 64-bit (sf == 1)

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then bits(16) imm = imm16;
integer pos; MoveWideOp opcode;

case opcode of
  when '00' opcode = MoveWideOp_N;
  when '10' opcode = MoveWideOp_Z;
  when '11' opcode = MoveWideOp_K;
  otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (wide immediate)</a>	! (IsZero(imm16) && hw != '00')

## Operation

```
bits(datasize) result;

result =if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
    result = NOT();

result<pos+15:pos> = imm16; {result};
X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

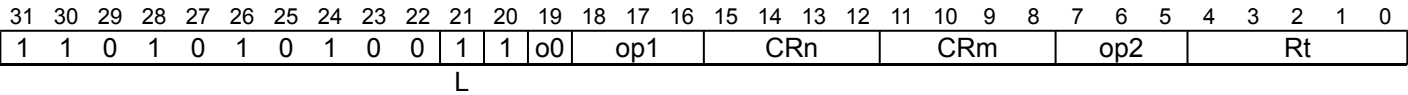
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from- [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

# MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.



## System

MRS <Xt>, (<systemreg>|S<op0>\_<op1>\_<Cn>\_<Cm>\_<op2>)

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm); {CRm};
boolean read = (L == '1');
```

## Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
- <systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".  
The System register names are defined in *AArch64 System Registers' in the System Register XML*.
- <op0> Is an unsigned immediate, encoded in "o0":

o0	<op0>
0	2
1	3
- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

## Operation

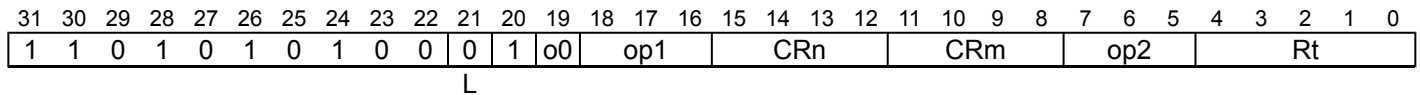
```
if read then
    X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t] = AArch64.SysRegRead(sys_op0,
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.



## System

MSR (<systemreg>|S<op0>\_<op1>\_<Cn>\_<Cm>\_<op2>), <Xt>

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);
```

```
integer t = UInt(Rt);
```

```
integer sys_op0 = 2 + UInt(o0);
```

```
integer sys_op1 = UInt(op1);
```

```
integer sys_op2 = UInt(op2);
```

```
integer sys_crn = UInt(CRn);
```

```
integer sys_crm = UInt(CRm); {CRm};
```

```
boolean read = (L == '1');
```

## Assembler Symbols

- <systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".  
The System register names are defined in *AArch64 System Registers' in the System Register XML*.
- <op0> Is an unsigned immediate, encoded in "o0":
- | o0 | <op0> |
|----|-------|
| 0  | 2     |
| 1  | 3     |
- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

## Operation

```
if read then
    X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

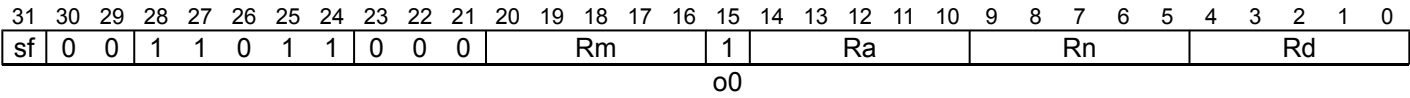
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.



MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias [MNEG](#).



32-bit (sf == 0)

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit (sf == 1)

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32; integer destsize = if sf == '1' then 64 else 32;
integer datasize = destsize;
boolean sub_op = (o0 == '1');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MNEG</a>	Ra == '11111'

## Operation

```
bits(destsize) operand1 =bits(datasize) operand1 = X[n];
bits(destsize) operand2 =bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
    result = UInt(operand3) - (UInt(operand1) * UInt(operand2));(operand2));
else
    result =
    UInt(operand3) + (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u><a href="#">ISA v82A A64 xml 00bet3.1</a></u>	htmldiff from-	<u><a href="#">(new)</a></u>
<u><a href="#">(old)</a></u>	ISA_v82A_A64_xml_00bet3.1	<u><a href="#">ISA v82A A64 xml 00bet3.1 OPT</a></u>

## MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M	Rm				1	0	0	0	H	0	Rn				Rd						

### Vector

MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>; product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	0	1	1	1	Rn						Rd			
U																															

### Three registers of the same type

MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

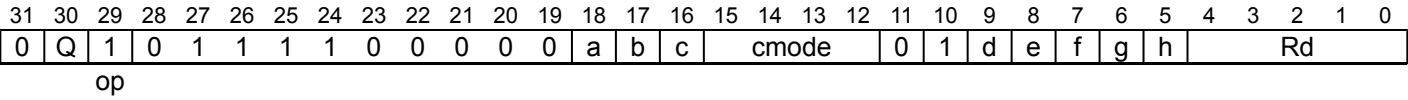
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

# MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## 16-bit shifted immediate (cmode == 10x0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

## 32-bit shifted immediate (cmode == 0xx0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

## 32-bit shifting ones (cmode == 110x)

```
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx01' operation = when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '1110x' operation = when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UnallocatedEncoding();
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff](#) from- [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)



## NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1
CRm																op2															

## System

NOP

```
// Empty.SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0010 000'
        op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;
    when '0010 001'
        op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;
    otherwise op = SystemHintOp_NOP;
```

## Operation

```
// do nothing
case op of
  when SystemHintOp_YIELD Hint_Yield();

  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, TRUE);
        if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
          // Check for traps described by the Hypervisor.
          AArch64.CheckForWFXTrap(EL2, TRUE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
          // Check for traps described by the Secure Monitor.
          AArch64.CheckForWFXTrap(EL3, TRUE);
        WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, FALSE);
        if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
          // Check for traps described by the Hypervisor.
          AArch64.CheckForWFXTrap(EL2, FALSE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
          // Check for traps described by the Secure Monitor.
          AArch64.CheckForWFXTrap(EL3, FALSE);
        WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBRegDomain_FullSystem, MBRegTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

  otherwise // do nothing
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	Rm					0	0	0	1	1	1	Rn					Rd				
size																															

### Three registers of the same type

ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean invert = (size<0> == '1'); LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

operand2 =if invert then operand2 = NOT(operand2);

result = operand1 OR operand2; case op of
when
LogicalOp_AND
result = operand1 AND operand2;
when LogicalOp_ORR
result = operand1 OR operand2;
V[d] = result;
```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias [MVN](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
sf		0	1	0		1	0	1	0	shift		1	Rm				imm6						Rn						Rd								
opc										N																											

### 32-bit (sf == 0)

ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then boolean setflags; LogicalOp op;
case op of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);(imm6);
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--

Alias Conditions

Alias	Is preferred when
<a href="#">MVN</a>	Rn == '11111'

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 =if invert then operand2 = NOT(operand2);

result = operand1 OR operand2;case op of
  when
LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp\_ORR result = operand1 OR operand2;
  when LogicalOp\_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
op												cmode																			

### 16-bit (cmode == 10x1)

ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}

### 32-bit (cmode == 0xx1)

ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx10' operation = when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '10x00' operation = when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x10' operation = when '10x01' operation = ImmediateOp_MVNI;
  when '10x10' operation = ImmediateOp_ORR;
  when '110x0' operation = when '10x11' operation = ImmediateOp_BIC;
  when '110x0' operation = ImmediateOp_MOVI;
  when '1110x' operation = when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UnallocatedEncoding;
  when '11110' operation = ();
  operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp MOVI
    result = imm;
  when ImmediateOp MVNI
    result = NOT(imm);
  when ImmediateOp ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1 \(old\)](#)      [html diff from-](#)      [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1    [ISA v82A A64 xml 00bet3.1 OPT](#)

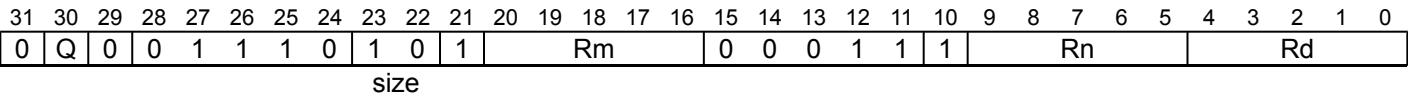


ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MOV \(vector\)](#).



Three registers of the same type

```
ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean invert = (size<0> == '1'); LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (vector)</a>	Rm == Rn

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = operand1 OR operand2; if invert then operand2 =
NOT(operand2);

case op of
  when LogicalOp_AND
    result = operand1 AND operand2;
  when LogicalOp_ORR
    result = operand1 OR operand2;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias [MOV \(bitmask immediate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	0	0	1	0	0	N	immr						imms						Rn				Rd					
opc																															

### 32-bit (sf == 0 && N == 0)

ORR <Wd|WSP>, <Wn>, #<imm>

### 64-bit (sf == 1)

ORR <Xd|SP>, <Xn>, #<imm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then boolean setflags; LogicalOp op;
case op of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;
bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (bitmask immediate)</a>	Rn == '11111' && ! <a href="#">MoveWidePreferred</a> (sf, N, imms, immr)

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

result = operand1 OR imm;
if d == 31 thencase op of
  when
    LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias [MOV \(register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		0		1		0		1		0		1		0		shift		0		Rm						imm6						Rn						Rd			
opc												N																													

### 32-bit (sf == 0)

ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then boolean setflags; LogicalOp op;
case op of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);(imm6);
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (register)</a>	shift == '00' && imm6 == '000000' && Rn == '11111'

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 OR operand2;if invert then operand2 =
NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

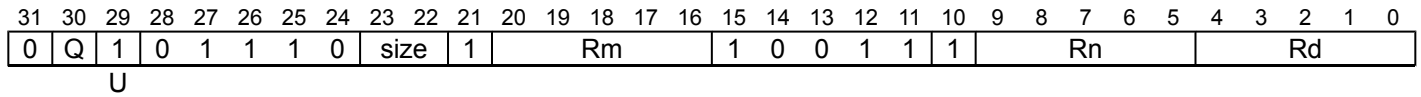
<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## PMUL

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



## PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#).

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
1		1		1		0		0		1		1		0		imm12												Rn						Rt			
size										opc																											

### Unsigned offset

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 3); {imm12, 64}, scale};
```

### Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.  
<type> is one of:

#### PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

#### PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

#### PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

#### L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

#### L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

#### L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

#### KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

#### STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see [Prefetch memory](#).

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.  
This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    address = if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset; if ! postindex then
    address = address + offset;

case memop of
    when
        MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(address, t<4:0>)[n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#).

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	imm19																			Rt				
opc																															

### Literal

PRFM (<prfop>|<imm5>), <label>

```
integer t = UInt(Rt);
bits(64) offset;

offset = (Rt); MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

### Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.  
<type> is one of:

#### PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

#### PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

#### PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

#### L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

#### L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

#### L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

**KEEP**

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

**STRM**

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5>	Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

**Operation**

```
bits(64) address = PC[] + offset;[] + offset;
bits(size*8) data;

case memop of
  when

MemOp_LOAD
  data = Mem[address, size, AccType_NORMAL];
  if signed then
    X[t] = SignExtend(data, 64);
  else
    X[t] = data;

  when MemOp_PREFETCHPrefetch(address, t<4:0>);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#).

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	Rm					option			S	1	0	Rn					Rt				
size					opc																										

## Integer

```
PRFM (<prfop>|<imm5>), [<Xn|SP>, (<Wm|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0; integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.  
<type> is one of:

### PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

### PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

### PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

### L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

### L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

### L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

### KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

### STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see [Prefetch memory](#).

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.  
This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":
option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX
<amount>	Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":
S	<amount>
0	#0
1	#3

Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;

```



## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    address = if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset; if ! postindex then
    address = address + offset;

case memop of
    when

MemOp_STORE
    if rt_unknown then
        data = bits(datasize) UNKNOWN;
    else
        data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(address, t<4:0>)[n] = address;
```

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## PRFM (unscaled offset)

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#).

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	0	imm9									0	0	Rn			Rt						
size											opc																				

### Unscaled offset

```
PRFUM (<prfop>|<imm5>), [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset =boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.  
<type> is one of:

#### PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

#### PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

#### PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

#### L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

#### L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

#### L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

#### KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

#### STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see [Prefetch memory](#).

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.  
This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    address = if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset; if ! postindex then
    address = address + offset;

case memop of
    when
        MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(address, t<4:0>)[n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA_v82A_A64_xml_00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA_v82A_A64_xml_00bet3.1 OPT</u></a>

## PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

### System (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	1	1	1	1	1	1
																CRm				op2											

### System

PSB CSYNC

```
SystemHintOp op;
```

```
op = ifcase CRm:op2 of
```

```
when '0000 000' op = SystemHintOp_NOP;
```

```
when '0000 001' op = SystemHintOp_YIELD;
```

```
when '0000 010' op = SystemHintOp_WFE;
```

```
when '0000 011' op = SystemHintOp_WFI;
```

```
when '0000 100' op = SystemHintOp_SEV;
```

```
when '0000 101' op = SystemHintOp_SEVL;
```

```
when '0010 000'
```

```
    op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;
```

```
when '0010 001'
```

```
    op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP; () then SystemHi
```

```
otherwise op = SystemHintOp_NOP;
```

## Operation

```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();

  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, TRUE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, TRUE);
      WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, FALSE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, FALSE);
      WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBRegDomain_FullSystem, MBRegTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

otherwise // do nothing
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

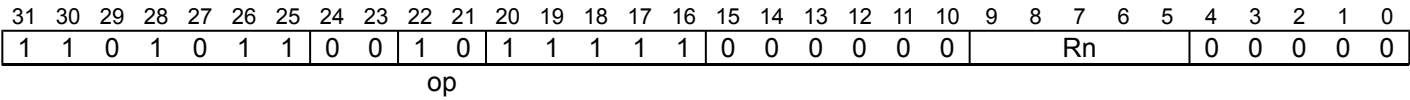
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.



Integer

```
RET {<Xn>}
```

```
integer n = UInt(Rn); BranchType branch_type;
case op of
  when '00' branch_type = BranchType_JMP;
  when '01' branch_type = BranchType_CALL;
  when '10' branch_type = BranchType_RET;
  otherwise UnallocatedEncoding();
```

Assembler Symbols

<Xn>      Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

Operation

```
bits(64) target = X[n];[n];
if branch_type ==
  BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_RET);(target, branch_type);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction [REV64](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	x	Rn						Rd							
																						opc													

### 32-bit (sf == 0 && opc == 10)

REV <Wd>, <Wn>

### 64-bit (sf == 1 && opc == 11)

REV <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UnallocatedEncoding();
    container_size = 64;
```

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        result<rev_index + 7:rev_index> = operand<index + 7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	0	1	1	0	Rn				Rd					
U																o0															

### Vector

REV16 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//   64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//   32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//   16+B = 2, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//   64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//   32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//   16+B = 1, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + {op} + UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```

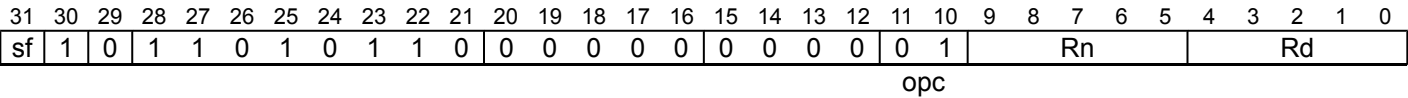
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

# REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.



## 32-bit (sf == 0)

```
REV16 <Wd>, <Wn>
```

## 64-bit (sf == 1)

```
REV16 <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UnallocatedEncoding();
    container_size = 64;
```

## Assembler Symbols

- <Wd>            Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>            Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd>            Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>            Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    result<rev_index + 7:rev_index> = operand<index + 7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#) [htmldiff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## REV32 (vector)

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn				Rd					
U										o0																					

### Vector

REV32 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//   64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//   32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//   16+B = 2, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//   64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//   32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//   16+B = 1, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + {op} + UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```

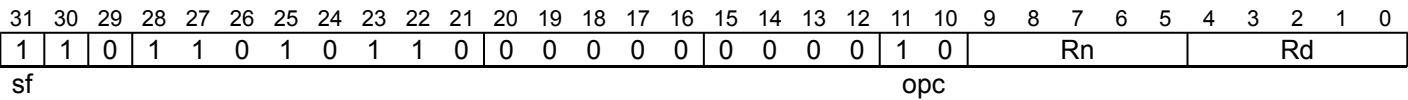
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.



### 64-bit

REV32 <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

### Assembler Symbols

- <Xd>                    Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>                    Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

### Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        result<rev_index + 7:rev_index> = operand<index + 7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## REV64

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn				Rd					
U										o0																					

### Vector

REV64 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize: B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
// 64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
// 32+B = 1, 32+H = 2, 32+S = X, 32+D = X
// 16+B = 2, 16+H = X, 16+S = X, 16+D = X
// 8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
// 64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
// 32+B = 2, 32+H = 1, 32+S = X, 32+D = X
// 16+B = 1, 16+H = X, 16+S = X, 16+D = X
// 8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + {op} + UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

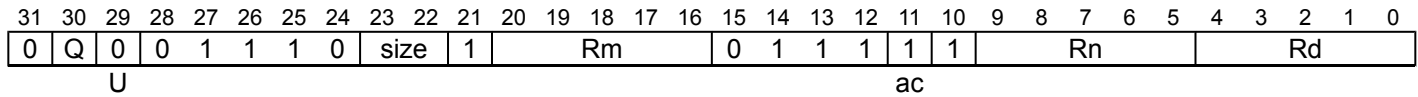
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## SABA

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>; (element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#) [html diff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register, while the SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	size	1	Rm						0	1	0	1	0	0	Rn						Rd					
U										op																							

### Three registers, not all the same type

SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>; (element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

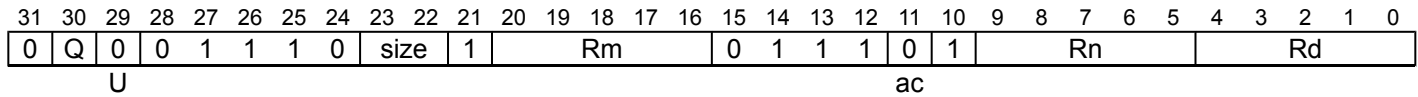
<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



## SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>; (element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#) [htmldiff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction writes the vector to the lower half of the destination register and clears the upper half, while the

SABDL2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	size	1	Rm						0	1	1	1	0	0	Rn						Rd					
U										op																							

### Three registers, not all the same type

SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>; (element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

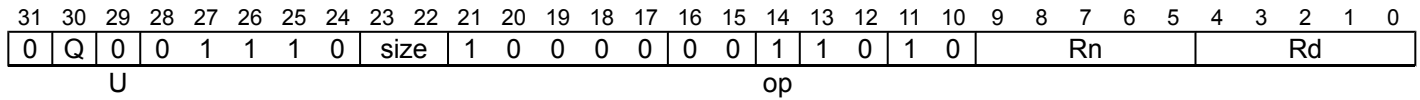
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Vector

SADALP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>; sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

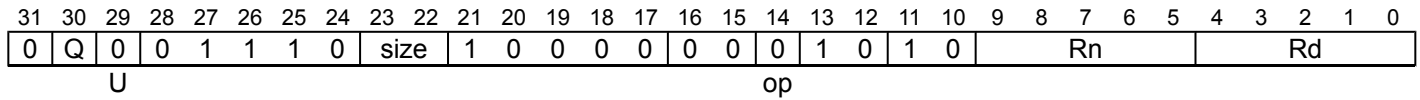
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u><a href="#">ISA v82A A64 xml 00bet3.1</a></u>	htmldiff from-	<u><a href="#">(new)</a></u>
<u><a href="#">(old)</a></u>	ISA_v82A_A64_xml_00bet3.1	<u><a href="#">ISA v82A A64 xml 00bet3.1 OPT</a></u>

## SADDLP

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Vector

SADDLP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>; sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u><a href="#">ISA v82A A64 xml 00bet3.1</a></u>	htmldiff from-	<u><a href="#">(new)</a></u>
<u><a href="#">(old)</a></u>	ISA_v82A_A64_xml_00bet3.1	<u><a href="#">ISA v82A A64 xml 00bet3.1 OPT</a></u>



## SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias [NGC](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

### 32-bit (sf == 0)

SBC <Wd>, <Wn>, <Wm>

### 64-bit (sf == 1)

SBC <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">NGC</a>	Rn == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzev;

operand2 =if sub_op then
    operand2 = NOT(operand2);

(result, -) = (result, nzev) = AddWithCarry(operand1, operand2, PSTATE.C); (operand1, operand2, PSTATE.C)
if setflags then
    PSTATE.<N,Z,C,V> = nzev;

X[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [ISA v82A A64 xml 00bet3.1 OPT](#)

## SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [NGCS](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

### 32-bit (sf == 0)

SBCS <Wd>, <Wn>, <Wm>

### 64-bit (sf == 1)

SBCS <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">NGCS</a>	Rn == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcvc;

operand2 =if sub_op then
    operand2 = NOT(operand2);

(result, nzcvc) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcvc;if setflags then
    PSTATE.<N,Z,C,V> = nzcvc;

X[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#)      `htmldiff from-`      [\(new\)](#)  
[\(old\)](#)      `ISA_v82A_A64_xml_00bet3.1`      [ISA v82A A64 xml 00bet3.1 OPT](#)

## SBFM

Signed Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, shifting in copies of the sign bit in the upper bits and zeros in the lower bits.

This instruction is used by the aliases [ASR \(immediate\)](#), [SBFIZ](#), [SBFX](#), [SXTB](#), [SXTH](#), and [SXTW](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0 0		1 0 0		1 1 0		N		immr						imms						Rn				Rd					
opc																															

### 32-bit (sf == 0 && N == 0)

SBFM <Wd>, <Wn>, #<immr>, #<imms>

### 64-bit (sf == 1 && N == 1)

SBFM <Xd>, <Xn>, #<immr>, #<imms>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then ease-ope-of
  when '00' inzero = TRUE; extend = TRUE; // SBFM
  when '01' inzero = FALSE; extend = FALSE; // BFM
  when '10' inzero = TRUE; extend = FALSE; // UBFM
  when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">ASR (immediate)</a>	32-bit	<code>imms == '011111'</code>
<a href="#">ASR (immediate)</a>	64-bit	<code>imms == '111111'</code>
<a href="#">SBFIZ</a>		<code>UInt(imms) &lt; UInt(immr)</code>
<a href="#">SBFX</a>		<code>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</code>
<a href="#">SXTB</a>		<code>immr == '000000' &amp;&amp; imms == '000111'</code>
<a href="#">SXTH</a>		<code>immr == '000000' &amp;&amp; imms == '001111'</code>
<a href="#">SXTW</a>		<code>immr == '000000' &amp;&amp; imms == '011111'</code>

Operation

```
bits(datasize) src =bits(datasize) dst = if inzero then Zeros() else X[d];  
bits(datasize) src = X[n];  
  
// perform bitfield move on low bits  
bits(datasize) bot = (dst AND NOT[n]);  
  
// perform bitfield move on low bits  
bits(datasize) bot =(wmask) OR ( ROR(src, R) AND wmask;  
(src, R) AND wmask);  
  
// determine extension bits (sign, zero or dest register)  
bits(datasize) top =bits(datasize) top = if extend then Replicate(src<S>;  
(src<S>) else dst;  
  
// combine extension bits and result bits  
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from-

[\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

### SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	0	0	0	0	1	0	scale				Rn				Rd								
										rmode		opcode																			

### 32-bit to half-precision (sf == 0 && type == 11) (ARMv8.2)

```
SCVTF <Hd>, <Wn>, #<fbits>
```

### 32-bit to single-precision (sf == 0 && type == 00)

```
SCVTF <Sd>, <Wn>, #<fbits>
```

### 32-bit to double-precision (sf == 0 && type == 01)

```
SCVTF <Dd>, <Wn>, #<fbits>
```

### 64-bit to half-precision (sf == 1 && type == 11) (ARMv8.2)

```
SCVTF <Hd>, <Xn>, #<fbits>
```

### 64-bit to single-precision (sf == 1 && type == 00)

```
SCVTF <Sd>, <Xn>, #<fbits>
```

### 64-bit to double-precision (sf == 1 && type == 01)

```
SCVTF <Dd>, <Xn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPCnvOp op;
FPRounding rounding;
boolean unsigned;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

rounding = case opcode<2:1>:rmode of
  when '00 11' // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPCnvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPCnvOp_CVT_ItoF;
  otherwise
    UnallocatedEncoding(FPCR); ();
```

## Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.



<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".  For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = case op of
  when FPCConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPCConvOp_CVT_ItoF
    intval = X[n];
fltval = FixedToFP(intval, fracbits, FALSE, FPCR, rounding);
V[d] = fltval;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	0	0	0	1	0	0	0	0	0	0	0	0	Rn				Rd					
										rmode		opcode																			

**32-bit to half-precision (sf == 0 && type == 11)**  
(ARMv8.2)

SCVTF <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && type == 00)**

SCVTF <Sd>, <Wn>

**32-bit to double-precision (sf == 0 && type == 01)**

SCVTF <Dd>, <Wn>

**64-bit to half-precision (sf == 1 && type == 11)**  
(ARMv8.2)

SCVTF <Hd>, <Xn>

**64-bit to single-precision (sf == 1 && type == 00)**

SCVTF <Sd>, <Xn>

**64-bit to double-precision (sf == 1 && type == 01)**

SCVTF <Dd>, <Xn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(FPCR); ();

```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[n];
  fltval = [d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, FALSE, FPCR, rounding); (intval, 0, unsigned, FPCR, rounding);
  V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n, part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = fltval; [d, part] = fltval;
```

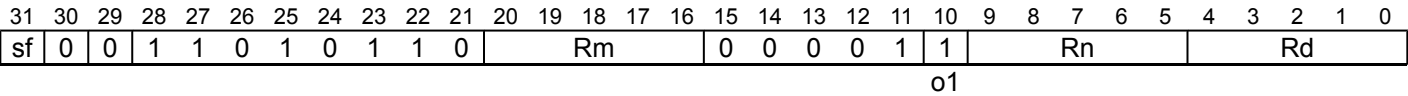
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

# SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.



## 32-bit (sf == 0)

```
SDIV <Wd>, <Wn>, <Wm>
```

## 64-bit (sf == 1)

```
SDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (o1 == '0');
```

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
  result = 0;
else
  result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real((operand1, unsigned)) / Real(Int(operand2, FALSE)));

X[d] = result<datasize-1:0>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see [Wait for Event mechanism and Send event](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	1	1	1
																CRm								op2							

## System

SEV

```
// Empty.SystemHintOp op;
case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0010 000'
    op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;
  when '0010 001'
    op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;
  otherwise op = SystemHintOp_NOP;
```

## Operation

```

case op of
  when SystemHintOp_YIELD
    Hint_Yield();

  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, TRUE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, TRUE);
      WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, FALSE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, FALSE);
      WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBRegDomain_FullSystem, MBRegTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();();

  otherwise // do nothing

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



## SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1	1
																CRm								op2							

## System

SEVL

```
// Empty.SystemHintOp op;
case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0010 000'
    op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;
  when '0010 001'
    op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;
  otherwise op = SystemHintOp_NOP;
```

## Operation

```

case op of
  when SystemHintOp_YIELD
    Hint_Yield();

  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, TRUE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, TRUE);
      WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, FALSE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, FALSE);
      WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBRegDomain_FullSystem, MBRegTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();();

  otherwise // do nothing

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

# SHA1C

SHA1 hash update (choose).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				

## Advanced SIMD

SHA1C <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

## Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckCryptoEnabled64();

bits(128) X = V[d];
bits(32) Y = V[n]; // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
  t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
  Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
  X<63:32> = ROL(X<63:32>, 30);
  <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# SHA1M

SHA1 hash update (majority).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	0	1	0	0	0	Rn					Rd				

## Advanced SIMD

SHA1M <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

## Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckCryptoEnabled64();

bits(128) X = V[d];
bits(32) Y = V[n]; // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
  t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
  Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
  X<63:32> = ROL(X<63:32>, 30);
  <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# SHA1P

SHA1 hash update (parity).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0			Rm			0	0	0	1	0	0			Rn					Rd		

## Advanced SIMD

SHA1P <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

## Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckCryptoEnabled64();

bits(128) X = V[d];
bits(32) Y = V[n]; // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
  t = SHAParity(X<63:32>, X<95:64>, X<127:96>);
  Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
  X<63:32> = ROL(X<63:32>, 30);
  <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# SHA1SU0

SHA1 schedule update 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	0	1	1	0	0	Rn					Rd				

## Advanced SIMD

SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;

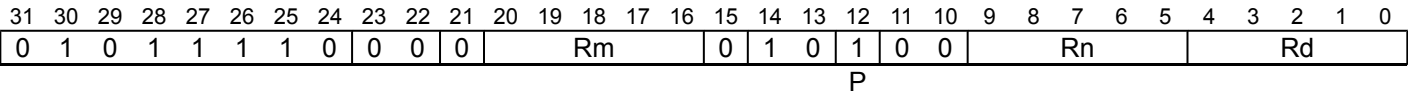
result = operand2<63:0>:operand1<127:64>;
result = operand2<63:0> :- operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# SHA256H2

SHA256 hash update (part 2).



## Advanced SIMD

SHA256H2 <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
boolean part1 = (P == '0');
```

## Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckCryptoEnabled64();

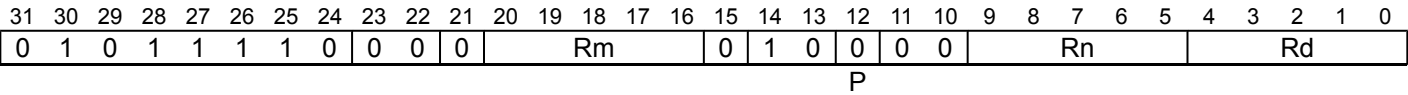
bits(128) result;
result =if part1 then
  result = SHA256hash(V[n], [d], V[d], [n], V[m], FALSE); [m], TRUE);
else
  result =
  SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# SHA256H

SHA256 hash update (part 1).



## Advanced SIMD

SHA256H <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
boolean part1 = (P == '0');
```

## Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckCryptoEnabled64();

bits(128) result;
result =if part1 then
  result = SHA256hash(V[d], V[n], V[m], TRUE);
else
  result =
  SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.



# SHA256SU0

SHA256 schedule update 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0										

## Advanced SIMD

```

SHA256SU0 <Vd>.4S, <Vn>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveCryptoExt() then UnallocatedEncoding();

```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0>:operand1<127:32>;
bits(128) T = operand2<31:0> : operand1<127:32>;
bits(32) elt;

for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;

```

## SHA256SU1

SHA256 schedule update 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0			Rm			0	1	1	0	0	0			Rn					Rd		

### Advanced SIMD

SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(128) T0 = operand3<31:0> : operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e-2, 32];
[T1, e-2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d] = result;
```

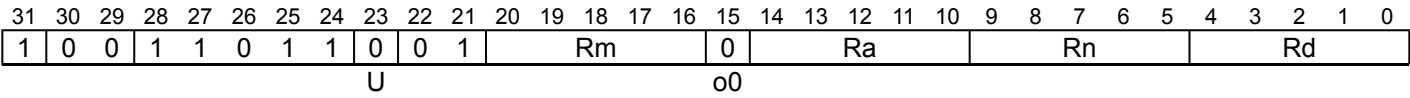
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [SMULL](#).



64-bit

```
SMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Xd>
- Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>
- Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm>
- Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa>
- Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">SMULL</a>	Ra == '11111'

Operation

```
bits(32) operand1 =bits(datasize) operand1 = X[n];
bits(32) operand2 =bits(datasize) operand2 = X[m];
bits(64) operand3 =bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
  result = Int(operand3, FALSE) + ((operand3, unsigned) - (Int(operand1, FALSE) * (operand1, unsigned)
else
  result =
Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

ISA v82A A64 xml 00bet3.1                      htmldiff from-                      (new)  
(old)                      ISA\_v82A\_A64\_xml\_00bet3.1    ISA v82A A64 xml 00bet3.1 OPT

## SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register, while the SMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm		0	0	1	0	H	0											
U										o2																					

### Vector

SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html](#)diff from- [\(new\)](#)  
[\(old\)](#) ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register, while the SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	0	0	0	0	Rn						Rd							
U										o1																									

### Three registers, not all the same type

SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from- [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)



## SMLS, SMLS2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLS instruction extracts vector elements from the lower half of the first source register, while the SMLS2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTL\\_EL2](#), and [CPTL\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm		0	1	1	0	H	0				Rn					Rd		
U										o2																					

### Vector

SMLS{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

ISA v82A A64 xml 00bet3.1 (old)      htldiff from-      (new)  
 ISA\_v82A\_A64\_xml\_00bet3.1      ISA v82A A64 xml 00bet3.1 OPT

## SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register, while the SMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	1	0	0	0	Rn						Rd							
U										o1																									

### Three registers, not all the same type

SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

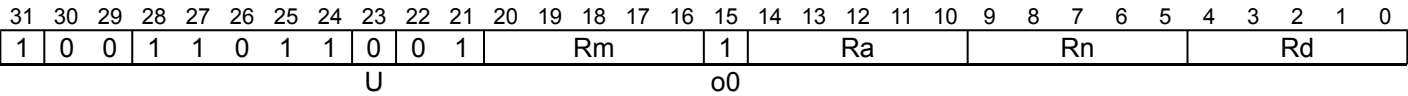
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

# SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [SMNEGL](#).



## 64-bit

SMSUBL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">SMNEGL</a>	Ra == '11111'

## Operation

```
bits(32) operand1 =bits(datasize) operand1 = X[n];
bits(32) operand2 =bits(datasize) operand2 = X[m];
bits(64) operand3 =bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
  result = Int(operand3, FALSE) - ((operand3, unsigned) - (Int(operand1, FALSE) * (operand1, unsigned))
else
  result =
Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

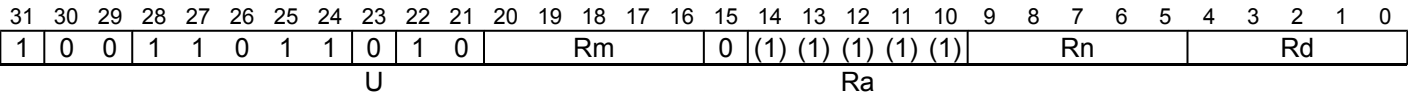
X[d] = result<63:0>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

# SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



## 64-bit

SMULH <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra); // ignored by UMULH/SMULH
integer destsize = 64;
integer datasize = destsize;
boolean unsigned = (U == '1');
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

```
bits(64) operand1 =bits(datasize) operand1 = X[n];
bits(64) operand2 =bits(datasize) operand2 = X[m];

integer result;

result = Int(operand1, FALSE) * (operand1, unsigned) * Int(operand2, FALSE); (operand2, unsigned);
X[d] = result<127:64>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register, while the SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm			1	0	1	0	H	0			Rn					Rd		
U																															

### Vector

SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts> [<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>; product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)



## SMULL, SMULL2 (vector)

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register, while the SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			1	1	0	0	0	0				Rn				Rd		
U																															

### Three registers, not all the same type

SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;[result, e, 2*esize] = (element1 * elem
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## SQDMLAL, SQDMLAL2 (by element)

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMLAL instruction extracts vector elements from the lower half of the first source register, while the

SQDMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M			Rm			0	0	1	1	H	0				Rn				Rd		
																	o2														

### Scalar

SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm			0	0	1	1	H	0				Rn				Rd		
																	o2														

Vector

SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

ISA v82A A64 xml 00bet3.1

htmldiff from-

(new)

(old)

ISA\_v82A\_A64\_xml\_00bet3.1

ISA v82A A64 xml 00bet3.1 OPT

## SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMLAL instruction extracts each source vector from the lower half of each source register, while the

SQDMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	1	1	1	0	size	1	Rm						1	0	0	1	0	0	Rn						Rd				
																o1																

### Scalar

SQDMLAL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	0	1	0	0	Rn						Rd					
o1																																	

### Vector

SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

## Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>



## SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMLSL instruction extracts vector elements from the lower half of the first source register, while the

SQDMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M			Rm			0	1	1	1	H	0				Rn				Rd		
																o2															

### Scalar

SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm			0	1	1	1	H	0				Rn				Rd		
																o2															

Vector

SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

ISA v82A A64 xml 00bet3.1

htmldiff from-

(new)

(old)

ISA\_v82A\_A64\_xml\_00bet3.1

ISA v82A A64 xml 00bet3.1 OPT

## SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMLSL instruction extracts each source vector from the lower half of each source register, while the

SQDMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	1	1	1	0	size	1	Rm						1	0	1	1	0	0	Rn						Rd				

o1

### Scalar

SQDMLSL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	1	1	0	0	Rn						Rd					

o1

### Vector

SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

## Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#) (new)

## SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

The SQDMULL instruction extracts the first source vector from the lower half of the first source register, while the

SQDMULL2 instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the *CPACR\_ELI*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M				Rm		1	0	1	1	H	0				Rn				Rd		

### Scalar

SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M	Rm				1	0	1	1	H	0	Rn				Rd						

Vector

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:



size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
```

```
bits(datasize) operand1 = Vpart[n, part];
```

```
bits(idxdsize) operand2 = V[m];
```

```
bits(2*datasize) result;
```

```
integer element1;
```

```
integer element2;
```

```
bits(2*esize) product;
```

```
boolean sat;
```

```
element2 = SInt(Elem[operand2, index, esize]);
```

```
for e = 0 to elements-1
```

```
    element1 = SInt(Elem[operand1, e, esize]);
```

```
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize); (2 * element1 * element2, 2*esize);
```

```
    Elem[result, e, 2*esize] = product;
```

```
    if sat then FPSR.QC = '1';
```

```
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff](#) from- [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register, while the

SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1				Rm			1	1	0	1	0	0				Rn				Rd		

### Scalar

SQDMULL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			1	1	0	1	0	0				Rn				Rd		

### Vector

SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

## Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize); {2 * element1 * element2, 2*esize};
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

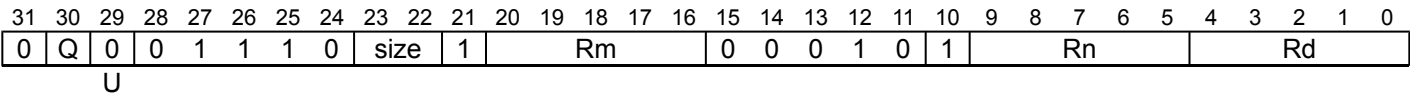
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *SHADD*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Three registers of the same type

```
SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>;[result, e, esize] = (element1 + element2 +
V[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

## ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	x	x	1	x	size		Rn				Rt					
L										opcode																					

### One register (opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>]
```

### Two registers (opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

### Three registers (opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

### Four registers (opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm				x	x	1	x	size			Rn				Rt					
L											opcode																				

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

- <Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();

```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	0	S	size	Rn					Rt					
L R										opcode																					

### 8-bit (opcode == 000)

```
ST1 { <Vt>.B } [<index>], [<Xn|SP>]
```

### 16-bit (opcode == 010 && size == x0)

```
ST1 { <Vt>.H } [<index>], [<Xn|SP>]
```

### 32-bit (opcode == 100 && size == 00)

```
ST1 { <Vt>.S } [<index>], [<Xn|SP>]
```

### 64-bit (opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D } [<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = integer UNKNOWN;  
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm					x	x	0	S	size	Rn					Rt					
L										R	opcode																				

#### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST1 { <Vt>.B } [<index>], [<Xn|SP>], #1
```

#### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST1 { <Vt>.B } [<index>], [<Xn|SP>], <Xm>
```

#### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H } [<index>], [<Xn|SP>], #2
```

#### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H } [<index>], [<Xn|SP>], <Xm>
```

#### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S } [<index>], [<Xn|SP>], #4
```

#### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S } [<index>], [<Xn|SP>], <Xm>
```

#### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D } [<index>], [<Xn|SP>], #8
```

#### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D } [<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<I>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	size		Rn				Rt					
L										opcode																					

### No offset

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm				1	0	0	0	size			Rn				Rt					
L											opcode																				

### Immediate offset (Rm == 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vt2>Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm>Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32
- <Xm>Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



## ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	0	S	size	Rn			Rt							
L										R	opcode																				

### 8-bit (opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B } [<index>], [<Xn|SP>]
```

### 16-bit (opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H } [<index>], [<Xn|SP>]
```

### 32-bit (opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S } [<index>], [<Xn|SP>]
```

### 64-bit (opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D } [<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	1	Rm				x	x	0	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B } [<index>], [<Xn|SP>], #2
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B } [<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H } [<index>], [<Xn|SP>], #4
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H } [<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S } [<index>], [<Xn|SP>], #8
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S } [<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D } [<index>], [<Xn|SP>], #16
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D } [<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<I>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	size	Rn				Rt						
L										opcode																					

### No offset

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm				0	1	0	0	size			Rn				Rt					
L											opcode																				

### Immediate offset (Rm == 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
 

Q	<imm>
0	#24
1	#48
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

### Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;      // number of iterations
integer selem;    // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;      // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;      // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;      // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;      // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;      // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;      // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;      // LD/ST1 (2 registers)
  otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	1	S	size	Rn			Rt							
L										R										opcode											

### 8-bit (opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>]

### 16-bit (opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>]

### 32-bit (opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>]

### 64-bit (opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm				x	x	1	S	size	Rn				Rt							
L										R	opcode																				



### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>], #3
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>], #6
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>], #12
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>], #24
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	size	Rn				Rt						
L										opcode																					

### No offset

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm				0	0	0	0	size	Rn				Rt							
L											opcode																				

### Immediate offset (Rm == 11111)

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
 

Q	<imm>
0	#32
1	#64
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;      // number of iterations
integer selem;    // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;      // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;      // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;      // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;      // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;      // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;      // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;      // LD/ST1 (2 registers)
  otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType\_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType\_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	1	S	size	Rn			Rt							
L										R	opcode																				

### 8-bit (opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

### 16-bit (opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

### 32-bit (opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

### 64-bit (opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	1	Rm				x	x	1	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.



## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UnallocatedEncoding();
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD has no memory ordering semantics.
- STADDL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	0	0	0	0	0	Rn					1	1	1	1	1
size		V				A									o3					opc											

#### 32-bit, no memory ordering (size == 10 && R == 0)

```
STADD <Ws>, [<Xn|SP>]
```

#### 32-bit, release (size == 10 && R == 1)

```
STADDL <Ws>, [<Xn|SP>]
```

#### 64-bit, no memory ordering (size == 11 && R == 0)

```
STADD <Xs>, [<Xn|SP>]
```

#### 64-bit, release (size == 11 && R == 1)

```
STADDL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then ldacctype = AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding;();
```

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, {address, datasize DIV 8, ldacctype}];
case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType_ATOMICRWMemAtomicOp_ADD];
result = value;

result = data + value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB has no memory ordering semantics.
- STADDLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	0	0	0	0	0	Rn					1	1	1	1	1
size		V					A										o3					opc									

#### No memory ordering (R == 0)

```
STADDB <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STADDLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, [address, datasize DIV 8, ldacctype];

case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType ATOMICRWMemAtomicOp_ADD];
result = value;

result = data + value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH has no memory ordering semantics.
- STADDLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1			Rs			0	0	0	0	0	0				Rn		1	1	1	1	1
size				V				A				o3				opc															

#### No memory ordering (R == 0)

```
STADDH <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STADDLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, [address, datasize DIV 8, ldacctype];

case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType ATOMICRWMemAtomicOp_ADD];
result = value;

result = data + value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>



## STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR has no memory ordering semantics.
- STCLRL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	0	0	1	0	0	Rn					1	1	1	1	1
size		V				A				o3					opc																

**32-bit, no memory ordering (size == 10 && R == 0)**

```
STCLR <Ws>, [<Xn|SP>]
```

**32-bit, release (size == 10 && R == 1)**

```
STCLRL <Ws>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && R == 0)**

```
STCLR <Xs>, [<Xn|SP>]
```

**64-bit, release (size == 11 && R == 1)**

```
STCLRL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then Idacctype = AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding;();
```

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, [address, datasize DIV 8, ldacctype]];

case op of
    when AccType ATOMICRW MemAtomicOp_ADD];
result = data AND result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP (value);
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRB has no memory ordering semantics.
- STCLRLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	0	0	1	0	0	Rn					1	1	1	1	1
size		V					A										o3					opc									

### No memory ordering (R == 0)

```
STCLRB <Ws>, [<Xn|SP>]
```

### Release (R == 1)

```
STCLRLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

## Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = data ANDresult = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value);
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## STCLR<sub>H</sub>, STCLR<sub>LH</sub>

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR<sub>H</sub> has no memory ordering semantics.
- STCLR<sub>LH</sub> stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1			Rs			0	0	0	1	0	0				Rn		1	1	1	1	1
size				V				A				o3				opc															

### No memory ordering (R == 0)

```
STCLRH <Ws>, [<Xn|SP>]
```

### Release (R == 1)

```
STCLRLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = data ANDresult = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP(value);
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEOR has no memory ordering semantics.
- STEORL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	0	1	0	0	0	Rn					1	1	1	1	1
size		V				A									o3					opc											

**32-bit, no memory ordering (size == 10 && R == 0)**

```
STEOR <Ws>, [<Xn|SP>]
```

**32-bit, release (size == 10 && R == 1)**

```
STEORL <Ws>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && R == 0)**

```
STEOR <Xs>, [<Xn|SP>]
```

**64-bit, release (size == 11 && R == 1)**

```
STEORL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then ldacctype = AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding;();
```

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, {address, datasize DIV 8, ldacctype}];

case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType_ATOMICRWMemAtomicOp_ADD];
result = value;

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



## STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORB has no memory ordering semantics.
- STEORLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
0	0	1	1	1	0	0	0	0	R	1	Rs					0	0	1	0	0	0	Rn					1	1	1	1	1					
size		V					A										o3					opc														

### No memory ordering (R == 0)

```
STEORB <Ws>, [<Xn|SP>]
```

### Release (R == 1)

```
STEORLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
  UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, [address, datasize DIV 8, ldacctype];

case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType ATOMICRWMemAtomicOp_ADD];
result = value;

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORH has no memory ordering semantics.
- STEORLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	0	1	0	0	0	Rn					1	1	1	1	1
size				V				A								o3				opc											

### No memory ordering (R == 0)

```
STEORH <Ws>, [<Xn|SP>]
```

### Release (R == 1)

```
STEORLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, [address, datasize DIV 8, ldacctype];

case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType ATOMICRWMemAtomicOp_ADD];
result = value;

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

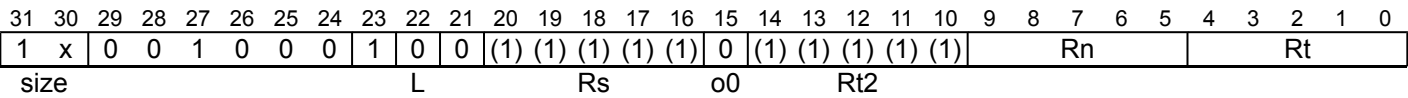
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

# STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

## No offset (ARMv8.1)



## 32-bit (size == 10)

```
STLLR <Wt>, [<Xn|SP>{, #0}]
```

## 64-bit (size == 11)

```
STLLR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << integer t2 = UInt(size); {Rt2}; // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt>                    Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>                    Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>                Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
when MemOp\_STORE
data = X[t];
Mem[address, dbytes, [address, dbytes, acctype] = data;

when
data = Mem[address, dbytes, acctype];
X[t] = ZeroExtendAccType LIMITEDORDEREDMemOp\_LOAD] = data; (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size								L				Rs				o0		Rt2													

### No offset

```
STLLRB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
    Mem[address, 1, [address, dbytes, acctype]] = data;

    when
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtendAccType_LIMITEDORDEREDMemOp_LOAD] = data; (data, regsize);
```

ISA v82A A64 xml 00bet3.1      htmldiff from-      (new)  
(old)      ISA\_v82A\_A64\_xml\_00bet3.1      ISA v82A A64 xml 00bet3.1 OPT



## STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size		L							Rs				o0		Rt2																

### No offset

STLLRH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
    Mem[address, 2, [address, dbytes, acctype]] = data;

    when
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtendAccType_LIMITEDORDEREDMemOp_LOAD] = data; (data, regsize);
```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L		Rs				o0		Rt2															

### 32-bit (size == 10)

STLR <Wt>, [<Xn|SP>{, #0}]

### 64-bit (size == 11)

STLR <Xt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << integer t2 = UInt(size); {Rt2}; // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
Mem[address, dbytes, {address, dbytes, acctype}] = data;

    when
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtendAccType_ORDEREDMemOp_LOAD = data; {data, regsize};
```

[ISA v82A A64 xml 00bet3.1](#) [htmldiff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L		Rs				o0		Rt2															

### No offset

STLRB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt); {Rt};
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
    Mem[address, 1, {address, dbytes, acctype}] = data;

    when
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtendAccType_ORDEREDMemOp_LOAD] = data; {data, regsize};
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire*, *Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size		L							Rs					o0		Rt2															

### No offset

STLRH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt); {Rt};
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
Mem[address, 2, {address, dbytes, acctype}] = data;

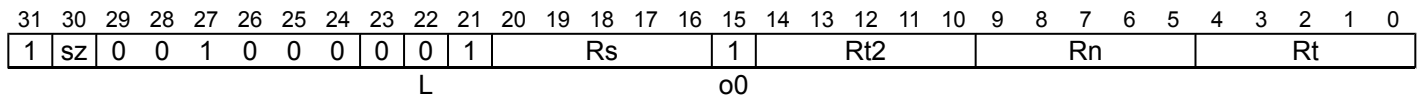
    when
        data = Mem[address, dbytes, acctype];
X[t] = ZeroExtendAccType_ORDEREDMemOp_LOAD] = data; {data, regsize};
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).



### 32-bit (sz == 0)

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release
integer elsize = 32 << (Rs); // ignored by all loads and store-release
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2; integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXP](#).

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.





```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (s == t2) then
  if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
      Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
      assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_NONE rt_unknown = FALSE; // store original value
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
      when Constraint_NONE rn_unknown = FALSE; // address is original base
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if n == 31 then
    CheckSPAlignment();
    address = SP[];
  elsif rn_unknown then
    address = bits(64) UNKNOWN;
  else
    address = X[n];

  if rt_unknown then
    data = bits(datasize) UNKNOWN;
  else
    bits(datasize DIV 2) el1 = case memop of
      when MemOp_STORE
        if rt_unknown then
          data = bits(datasize) UNKNOWN;
        elsif pair then
          bits(datasize DIV 2) el1 = X[t];
          bits(datasize DIV 2) el2 = X[t2];
          data = if BigEndian() then el1:el2 else el2:el1;
    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if() then el1 : el2 else el2 : el1;
    else
      data = X[t];

    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
      // This atomic write will be rejected if it does not refer
      // to the same physical locations after address translation.
      Mem[address, dbytes, [address, dbytes, acctype]] = data;
      status = AccType.ORDEREDExclusiveMonitorsStatus = data;
      status = (); [s] = ZeroExtend(status, 32);

```

```

when MemOp_LOAD
    // Tell the Exclusive Monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusive Monitor will only be set if all the reads are from the
    // same dbytes-aligned physical address, to allow for the possibility of
    // an atomicity break if the translation is changed between reads.
    AArch64.SetExclusiveMonitors(address, dbytes);

    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
        elseif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acetype];
            if BigEndian() then
                X[t] = data<datasize-1:elsize>;
                X[t2] = data<elsize-1:0>;
            else
                X[t] = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acetype, iswrite, secondstage));
                X[t] = Mem[address + 0, 8, acetype];
                X[t2] = Mem[address + 8, 8, acetype];
            else
                data = MemExclusiveMonitorsStatusX();[address, dbytes, acetype];
                X[s] = {t} = ZeroExtend(status, 32);(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size				L				o0				Rt2																			

### 32-bit (size == 10)

```
STLXR <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
STLXR <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release

integer elsize = 8 << (Rt2); // ignored by load/store single register
integer s = UInt(size); (Rs); // ignored by all loads and store-release
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXR](#).

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then if memop ==
  MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
      when Constraint_NONE rt_unknown = FALSE; // store original value
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if s == n && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
    when Constraint_NONE rn_unknown = FALSE; // address is original base
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elseif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

if rt_unknown then
  data = bits(elsize) UNKNOWN;
else
  data = case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      elsif pair then
        bits(datasize DIV 2) el1 = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if bits(datasize DIV 2) el2 = X[t2];
  data = if BigEndian() then el1 : el2 else el2 : el1;
else
  data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
  // This atomic write will be rejected if it does not refer
  // to the same physical locations after address translation.
  Mem[address, dbytes, [address, dbytes, acctype] = data;

```

```

        status = AccType_ORDEREDExclusiveMonitorsStatus] = data;
    status = (); [s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acetype];
            if BigEndian() then
                X[t] = data<datasize-1:elsize>;
                X[t2] = data<elsize-1:0>;
            else
                X[t] = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acetype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acetype];
                    X[t2] = Mem[address + 8, 8, acetype];
                else
                    data = MemExclusiveMonitorsStatusX() [address, dbytes, acetype];
X[s] = [t] = ZeroExtend(status, 32); (data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-

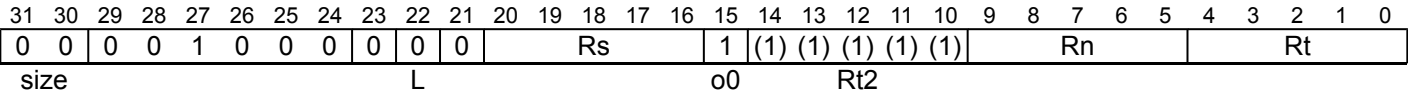
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

# STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).



## No offset

```
STLXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release(Rt2); // ignored by load
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRB](#).

## Assembler Symbols

- <Ws>

0

1

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

If the operation updates memory.

If the operation fails to update memory.
- <Wt>

<Xn|SP>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.





```

bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NONE rn_unknown = FALSE; // address is original base
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1 : el2 else el2 : el1;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation. (address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.

```

```

Mem[address, 1, [address, dbytes, acctype] = data;
status = AccType ORDEREDExclusiveMonitorsStatus] = data;
status = (); [s] = ZeroExtend(status, 32);

when MemOp_LOAD
    // Tell the Exclusive Monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusive Monitor will only be set if all the reads are from the
    // same dbytes-aligned physical address, to allow for the possibility of
    // an atomicity break if the translation is changed between reads.
    AArch64.SetExclusiveMonitors(address, dbytes);

    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian() then
                X[t] = data<datasize-1:elsize>;
                X[t2] = data<elsize-1:0>;
            else
                X[t] = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = MemExclusiveMonitorsStatusX(); [address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32); (data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size								L				o0				Rt2															

### No offset

STLXRH <Ws>, <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release (Rt2); // ignored by load
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRH](#).

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NONE rn_unknown = FALSE; // address is original base
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1 : el2 else el2 : el1;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation. (address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.

```

```

Mem[address, 2, [address, dbytes, acctype] = data;
status = AccType ORDEREDExclusiveMonitorsStatus] = data;
status = (); [s] = ZeroExtend(status, 32);

when MemOp_LOAD
    // Tell the Exclusive Monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusive Monitor will only be set if all the reads are from the
    // same dbytes-aligned physical address, to allow for the possibility of
    // an atomicity break if the translation is changed between reads.
    AArch64.SetExclusiveMonitors(address, dbytes);

    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian() then
                X[t] = data<datasize-1:elsize>;
                X[t2] = data<elsize-1:0>;
            else
                X[t] = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = MemExclusiveMonitorsStatusX(); [address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32); (data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1 0 1		1 0 0 0		0		imm7								Rt2				Rn				Rt							
L																															

### 32-bit (opc == 00)

```
STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.boolean wback = FALSE;  
boolean postindex = FALSE;
```

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
if opc == '11' then (Rt2); AccType acctype = AccType_VECSTREAM;  
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
if opc == '11' then UnallocatedEncoding();  
integer scale = 2 + UInt(opc);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop == MemOp_LOAD && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN then rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF then UnallocatedEncoding();
            when Constraint_NOP then EndOfInstruction();
    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    address = address + offset;
    if ! postindex then
        address = address + offset;

    data1 = case memop of
        when MemOp_STORE then
            data1 = V[t];
    data2 = V[t2];
    Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_VECSTREAMMem = data1; [address + 0, dbytes, acctype] = data2;
    when MemOp_LOAD then
        data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
        data2 = [address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t] = data1;
        V[t2] = data2;

    if wback then
        if postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            XAccType_VECSTREAMMem = data2; [n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#) (new)

## STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see [Load/Store addressing modes](#). For information about Non-temporal pair instructions, see [Load/Store Non-temporal pair](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
x		0		1		0		1		0		0		0		0		0		imm7							Rt2				Rn				Rt			
opc										L																												

### 32-bit (opc == 00)

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 10)

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.boolean wback = FALSE;
boolean postindex = FALSE;
```

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then(Rt2); AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data1 = case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
data2 = if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
    else
    data2 = X[t2];
Mem[address, dbytes, {address + 0, dbytes, acctype}] = data1; AccType_STREAMMem = data1; {address +

    when
MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, {address + 0, dbytes, acctype}];
    data2 = {address + dbytes, dbytes, acctype};
    if rt_unknown then
        data1 = bits(datasize) UNKNOWN;
        data2 = bits(datasize) UNKNOWN;
    X[t] = data1;
    X[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_STREAMMem = data2; [n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	(new)
(old)	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
opc		1		0		1		1		0		0		1		0		imm7							Rt2					Rn					Rt			
L																																						

#### 32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>

#### 64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

#### 128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

```
boolean wback = TRUE;  
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1 0 1		1		0 1 1		0		imm7							Rt2					Rn					Rt				
										L																					

#### 32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>, #<imm>]!

#### 64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!

#### 128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!

```
boolean wback = TRUE;  
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	0	0	imm7							Rt2					Rn					Rt				

### 32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;
```

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	<p>For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as &lt;imm&gt;/16.</p> <p>For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/16.</p>

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
if opc == '11' then (Rt2); AccType acctype = AccType_VEC;  
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
if opc == '11' then UnallocatedEncoding();  
integer scale = 2 + UInt(opc);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop == MemOp_LOAD && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN then rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF then UnallocatedEncoding();
            when Constraint_NOP then EndOfInstruction();
    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    if !postindex then
        if !postindex then
            address = address + offset;

    data1 = case memop of
        when MemOp_STORE then
            data1 = V[t];
    data2 = V[t2];
    Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_VECMem = data1; [address + dby
    when
    MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
        data2 = [address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t] = data1;
        VAccType_VECMem = data2;
    [t2] = data2;

    if wback then
        if postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#) (new)

## STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	0	1	0	1	0	0	0	1	0	imm7							Rt2				Rn				Rt						
opc										L																					

#### 32-bit (opc == 00)

STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>

#### 64-bit (opc == 10)

STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

```
boolean wback = TRUE;  
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
x		0	1		0	1	0	0		1	1	0	imm7							Rt2				Rn				Rt							
opc										L																									

#### 32-bit (opc == 00)

STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!

#### 64-bit (opc == 10)

STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!

```
boolean wback = TRUE;  
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
x		0	1		0	1		0	0		1	0	0		imm7							Rt2				Rn				Rt			
opc										L																							

### 32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	<p>For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/8.</p>

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
if L:opc<0> == '01' || opc == '11' then (Rt2); AccType acctype = AccType_NORMAL;  
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();  
boolean signed = (opc<0> != '0');  
integer scale = 2 + UInt(opc<1>);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```





```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
  if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
      when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is pre-writeback
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  if !postindex then
    address = address + offset;

if rt_unknown && t == n then
  data1 = bits(datasize) UNKNOWN;
else
  data1 = case memop of
    when MemOp_STORE
      if rt_unknown && t == n then
        data1 = bits(datasize) UNKNOWN;
      else
        data1 = X[t];
  if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
  else
    data2 = X[t2];
Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_NORMALMem] = data1; [address +
  when
MemOp_LOAD
  data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
  data2 = [address + dbytes, dbytes, acctype];
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
  if signed then
    X[t] = SignExtend(data1, 64);
    X[t2] = SignExtend(data2, 64);

```

```

else
    X[t] = data1;
    XAccType NORMALMem] = data2;
[t2] = data2;

if wback then
    if postindex then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elseif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9									0	1	Rn				Rt					
opc																															

#### 8-bit (size == 00 && opc == 00)

STR <Bt>, [<Xn|SP>], #<sim>

#### 16-bit (size == 01 && opc == 00)

STR <Ht>, [<Xn|SP>], #<sim>

#### 32-bit (size == 10 && opc == 00)

STR <St>, [<Xn|SP>], #<sim>

#### 64-bit (size == 11 && opc == 00)

STR <Dt>, [<Xn|SP>], #<sim>

#### 128-bit (size == 00 && opc == 10)

STR <Qt>, [<Xn|SP>], #<sim>

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9									1	1	Rn				Rt					
opc																															

8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>, #<sim>]!
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, #<sim>]!
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, #<sim>]!
```

64-bit (size == 11 && opc == 00)

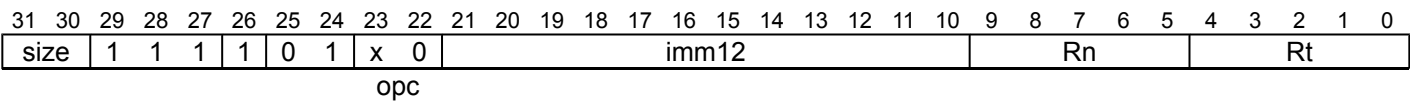
```
STR <Dt>, [<Xn|SP>, #<sim>]!
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	<p>For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.</p> <p>For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/2.</p> <p>For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/4.</p> <p>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/8.</p> <p>For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/16.</p>

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

[ISA v82A A64 xml 00bet3.1](#) [html diff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn				Rt					
size											opc																				

#### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn				Rt					
size											opc																				

#### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1		x		1		1		1		0		0		1		0		0		imm12												Rn				Rt			
size											opc																												



### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.  For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale; (Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if ope<1> == '0' then
    // store or zero-extending load
    memop = if ope<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && ope<0> == '1' then UnallocatedEncoding();
        regsize = if ope<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
  c = if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then if n == 31 then
  if memop !=
    MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
if ! postindex then
  address = address + offset;

if rt_unknown then
  data = bits(datasize) UNKNOWN;
else
  data = case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

  when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCHPrefetchAccType NORMALMemOp_LOAD] = data;
(address, t<4:0>);

if wback then
  if postindex then
    if wb_unknown then
      address = bits(64) UNKNOWN;
    elsif postindex then
      address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

```

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	1	0	0	x	0	1																						
opc																	Rm			option			S	1	0	Rn			Rt		

### 8-bit (size == 00 && opc == 00 && option != 011)

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 8-bit (size == 00 && opc == 00 && option == 011)

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### 16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SCTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt (Rn) ;
integer t = UInt (Rt) ;
integer m = UInt (Rm) ;
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if !postindex then
    address = address + offset;

case memop of
    when MemOp\_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp\_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	0	1	Rm				option			S	1	0	Rn				Rt						
size										opc																					

### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
if option<1> == '0' then UnallocatedEncoding(); // sub-word index  
ExtendType extend_type = DecodeRegExtend(option);  
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount>	For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":
----------	---

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale; (Rm); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```



## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

    when
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_NORMALMemOp_LOAD = data; [n] = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

## STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn				Rt					
size											opc																				

### Post-index

```
STRB <Wt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn				Rt					
size											opc																				

### Pre-index

```
STRB <Wt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0		0		1		1		1		0		0		1		0		0		imm12											Rn					Rt			
size											opc																												

### Unsigned offset

```
STRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0); (imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STRB \(immediate\)](#).

## Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if ope<1> == '0' then
    // store or zero-extending load
    memop = if ope<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && ope<0> == '1' then UnallocatedEncoding();
        regsize = if ope<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;

```

## Operation

```

bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
  c = if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPL);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then if n == 31 then
  if memop !=
    MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
if ! postindex then
  address = address + offset;

if rt_unknown then
  data = bits(8) UNKNOWN;
else
  data = case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

  when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCHPrefetchAccType_NORMALMemOp_LOAD] = data;
(address, t<4:0>);

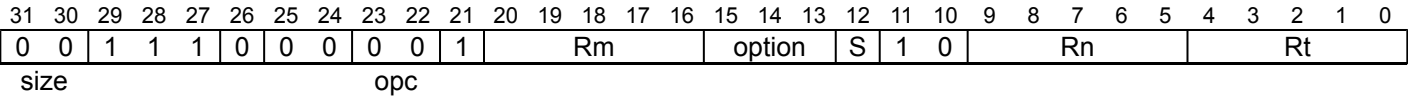
if wback then
  if postindex then
    if wb_unknown then
      address = bits(64) UNKNOWN;
    elsif postindex then
      address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

```

[ISA v82A A64 xml 00bet3.1](#)      [html](#)diff from-      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see [Load/Store addressing modes](#).  
The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



Extended register (option != 011)

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- <Wt>
- Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>
- Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>
- When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>
- When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>
- Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

- <amount>
- Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm); Acctype acctype = Acctype_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```



## Operation

```

bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    address = address + offset;
    if ! postindex then
        address = address + offset;

    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
    Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

    when
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            XAccType NORMAL MemOp_LOAD = data; [n] = address;

```



## STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn				Rt									
size										opc																									

### Post-index

```
STRH <Wt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn				Rt									
size										opc																									

### Pre-index

```
STRH <Wt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
0		1		1		1		0		0		1		0		0		imm12											Rn				Rt			
size										opc																										

### Unsigned offset

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1); (imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STRH \(immediate\)](#).

## Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;

```

## Operation

```

bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
  c = if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then if n == 31 then
  if memop !=
    MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
if ! postindex then
  address = address + offset;

if rt_unknown then
  data = bits(16) UNKNOWN;
else
  data = case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

  when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCHPrefetchAccType_NORMALMemOp_LOAD] = data;
(address, t<4:0>);

if wback then
  if postindex then
    if wb_unknown then
      address = bits(64) UNKNOWN;
    elsif postindex then
      address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

```

[ISA v82A A64 xml 00bet3.1](#)      [html](#)diff from-      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

## STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see [Load/Store addressing modes](#).

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	1	1	0	0	0	0	0	1	Rm					option			S	1	0	Rn				Rt						
size											opc																					

### 32-bit

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0; integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount>	Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":
----------	---

S	<amount>
0	#0
1	#1

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm); Acctype acctype = Acctype_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```



## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

    when
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_NORMAL MemOp_LOAD = data; [n] = address;

```



## STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSET has no memory ordering semantics.
- STSETL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	0	1	1	0	0	Rn					1	1	1	1	1
size		V				A									o3					opc											

**32-bit, no memory ordering (size == 10 && R == 0)**

```
STSET <Ws>, [<Xn|SP>]
```

**32-bit, release (size == 10 && R == 1)**

```
STSETL <Ws>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && R == 0)**

```
STSET <Xs>, [<Xn|SP>]
```

**64-bit, release (size == 11 && R == 1)**

```
STSETL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then ldacctype = AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding;();
```

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, {address, datasize DIV 8, ldacctype}];
case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType_ATOMICRWMemAtomicOp_ADD];
result = value;

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB has no memory ordering semantics.
- STSETLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	0	1	1	0	0	Rn					1	1	1	1	1
size		V					A										o3					opc									

#### No memory ordering (R == 0)

```
STSETB <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STSETLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, [address, datasize DIV 8, ldacctype];];

case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType ATOMICRWMemAtomicOp_ADD];
result = value;

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH has no memory ordering semantics.
- STSETLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1			Rs			0	0	1	1	0	0			Rn			1	1	1	1	1
size				V				A				o3				opc															

### No memory ordering (R == 0)

```
STSETH <Ws>, [<Xn|SP>]
```

### Release (R == 1)

```
STSETLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 <<
  UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, [address, datasize DIV 8, ldacctype];

case op of
    when result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWPAccType ATOMICRWMemAtomicOp_ADD];
result = value;

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>



## STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX has no memory ordering semantics.
- STSMAXL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs				0	1	0	0	0	0	Rn				1	1	1	1	1		
size		V				A								o3				opc													

#### 32-bit, no memory ordering (size == 10 && R == 0)

```
STSMAX <Ws>, [<Xn|SP>]
```

#### 32-bit, release (size == 10 && R == 1)

```
STSMAXL <Ws>, [<Xn|SP>]
```

#### 64-bit, no memory ordering (size == 11 && R == 0)

```
STSMAX <Xs>, [<Xn|SP>]
```

#### 64-bit, release (size == 11 && R == 1)

```
STSMAXL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then ldacctype = AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding;();
```

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, [address, datasize DIV 8, ldacctype]];

case op of
    when AccType ATOMICRW MemAtomicOp_ADD];
result = if result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STSMAXB, STSMAXB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXB has no memory ordering semantics.
- STSMAXB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	1	0	0	0	0	Rn				1	1	1	1	1		
size				V			A						o3			opc															

#### No memory ordering (R == 0)

```
STSMAXB <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STSMAXB <Ws>, [<Xn|SP>]
```

```

if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();

```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = ifresult = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

# STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXH has no memory ordering semantics.
- STSMAXLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

## Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1			Rs			0	1	0	0	0	0				Rn		1	1	1	1	1
size				V				A				o3				opc															

### No memory ordering (R == 0)

```
STSMAXH <Ws>, [<Xn|SP>]
```

### Release (R == 1)

```
STSMAXLH <Ws>, [<Xn|SP>]
```

```

if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);(Rs);

integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding;();

```

## Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = ifresult = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMIN has no memory ordering semantics.
- STSMINL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	1	0	1	0	0	Rn					1	1	1	1	1
size		V				A									o3					opc											

**32-bit, no memory ordering (size == 10 && R == 0)**

```
STSMIN <Ws>, [<Xn|SP>]
```

**32-bit, release (size == 10 && R == 1)**

```
STSMINL <Ws>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && R == 0)**

```
STSMIN <Xs>, [<Xn|SP>]
```

**64-bit, release (size == 11 && R == 1)**

```
STSMINL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then ldacctype = AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding;();
```

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, {address, datasize DIV 8, ldacctype}];

case op of
    when AccType ATOMICRW MemAtomicOp_ADD];
result = if result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP (value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



## STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINB has no memory ordering semantics.
- STSMINLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	1	0	1	0	0	Rn				1	1	1	1	1		
size				V			A						o3			opc															

#### No memory ordering (R == 0)

```
STSMINB <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STSMINLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = ifresult = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    (value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINH has no memory ordering semantics.
- STSMINLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1			Rs			0	1	0	1	0	0				Rn		1	1	1	1	1
size				V				A				o3				opc															

#### No memory ordering (R == 0)

```
STSMINH <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STSMINLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = ifresult = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP   (value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	0	0	imm9									1	0	Rn				Rt									
size										opc																									

### 32-bit (size == 10)

```
STTR <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (size == 11)

```
STTR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale; (Rt); AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_UNPRIVMemOp_LOAD = data; {n} = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

ISA v82A A64 xml 00bet3.1  
(old)

htmldiff from- (new)  
ISA\_v82A\_A64\_xml\_00bet3.1 ISA v82A A64 xml 00bet3.1 OPT

## STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	0	0	0	imm9									1	0	Rn				Rt									
size										opc																									

### Unscaled offset

```
STTRB <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```



## Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_UNPRIVMemOp_LOAD = data; [n] = address;
```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR\_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	0	0	imm9									1	0	Rn				Rt					
size											opc																				

### Unscaled offset

```
STTRH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt); AccType acctype = AccType_UNPRIV;  
MemOp memop;  
boolean signed;  
integer regsize;  
  
if opc<1> == '0' then  
    // store or zero-extending load  
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;  
    regsize = if size == '11' then 64 else 32;  
    signed = FALSE;  
else  
    if size == '11' then  
        UnallocatedEncoding();  
    else  
        // sign-extending load  
        memop = MemOp_LOAD;  
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();  
        regsize = if opc<0> == '1' then 32 else 64;  
        signed = TRUE;  
    end  
integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_UNPRIVMemOp_LOAD = data; [n] = address;
```

ISA v82A A64 xml 00bet3.1      htmldiff from-      (new)  
(old)      ISA\_v82A\_A64\_xml\_00bet3.1      ISA v82A A64 xml 00bet3.1 OPT

## STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAX has no memory ordering semantics.
- STUMAXL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	1	1	0	0	0	Rn					1	1	1	1	1
size		V				A									o3					opc											

**32-bit, no memory ordering (size == 10 && R == 0)**

```
STUMAX <Ws>, [<Xn|SP>]
```

**32-bit, release (size == 10 && R == 1)**

```
STUMAXL <Ws>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && R == 0)**

```
STUMAX <Xs>, [<Xn|SP>]
```

**64-bit, release (size == 11 && R == 1)**

```
STUMAXL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then ldacctype = AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding;();
```

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, [address, datasize DIV 8, ldacctype];

case op of
    when AccType ATOMICRW MemAtomicOp_ADD];

result = if result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## STUMAXB, STUMAXB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXB has no memory ordering semantics.
- STUMAXB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	1	1	0	0	0	Rn					1	1	1	1	1
size		V					A										o3					opc									

### No memory ordering (R == 0)

STUMAXB <Ws>, [<Xn|SP>]

### Release (R == 1)

STUMAXB <Ws>, [<Xn|SP>]

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = ifresult = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH has no memory ordering semantics.
- STUMAXLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	1	1	0	0	0	Rn					1	1	1	1	1
size		V					A										o3					opc									

#### No memory ordering (R == 0)

```
STUMAXH <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STUMAXLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = ifresult = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    (value) then data else value;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMIN has no memory ordering semantics.
- STUMINL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	1	1	1	0	0	Rn					1	1	1	1	1
size		V				A									o3					opc											

**32-bit, no memory ordering (size == 10 && R == 0)**

```
STUMIN <Ws>, [<Xn|SP>]
```

**32-bit, release (size == 10 && R == 1)**

```
STUMINL <Ws>, [<Xn|SP>]
```

**64-bit, no memory ordering (size == 11 && R == 0)**

```
STUMIN <Xs>, [<Xn|SP>]
```

**64-bit, release (size == 11 && R == 1)**

```
STUMINL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then ldacctype = AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding;();
```

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, [address, datasize DIV 8, ldacctype]];

case op of
    when AccType ATOMICRW MemAtomicOp_ADD];
result = if result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP (value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINB has no memory ordering semantics.
- STUMINLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	1	1	1	0	0	Rn					1	1	1	1	1
size		V					A										o3					opc									

#### No memory ordering (R == 0)

```
STUMINB <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STUMINLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = ifresult = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    (value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)

[ISA v82A A64 xml 00bet3.1 OPT](#)

## STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINH has no memory ordering semantics.
- STUMINLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1			Rs			0	1	1	1	0	0				Rn		1	1	1	1	1
size				V				A				o3				opc															

#### No memory ordering (R == 0)

```
STUMINH <Ws>, [<Xn|SP>]
```

#### Release (R == 1)

```
STUMINLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType stacctype = if R == '1' then idacctype = AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, [address, datasize DIV 8, ldacctype];

case op of
    when AccType_ATOMICRWMemAtomicOp_ADD];
result = ifresult = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP   (value) then value else data;
result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result; [address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9									0	0	Rn				Rt					
opc																															

### 8-bit (size == 00 && opc == 00)

```
STUR <Bt>, [<Xn|SP>{, #<sim>}]
```

### 16-bit (size == 01 && opc == 00)

```
STUR <Ht>, [<Xn|SP>{, #<sim>}]
```

### 32-bit (size == 10 && opc == 00)

```
STUR <St>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (size == 11 && opc == 00)

```
STUR <Dt>, [<Xn|SP>{, #<sim>}]
```

### 128-bit (size == 00 && opc == 10)

```
STUR <Qt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

case memop of
    when MemOp\_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp\_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

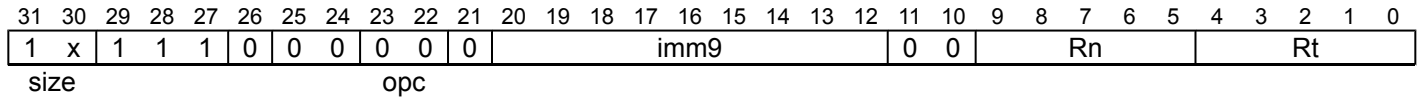
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes*.



### 32-bit (size == 10)

```
STUR <Wt>, [<Xn|SP>{, #<simm>}]
```

### 64-bit (size == 11)

```
STUR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale; (Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_NORMAL MemOp_LOAD = data; {n} = address;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from- (new)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
0		0		1		1		1		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0	

### Unscaled offset

```
STURB <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

when MemOp_PREFETCHPrefetch(address, t<4:0>);

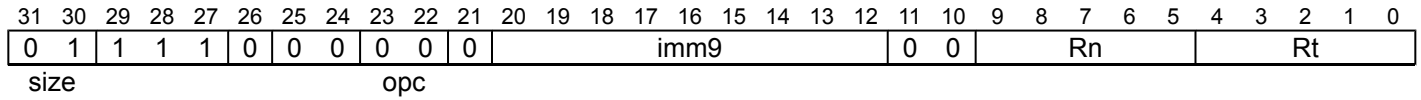
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_NORMALMemOp_LOAD = data; [n] = address;
```

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)



## STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see [Load/Store addressing modes](#).



### Unscaled offset

```
STURH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt); AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then if memop ==
    MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
        X[t] = SignExtend(data, regsize);
    else
        X[t] = ZeroExtend(data, regsize);

when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_NORMALMemOp_LOAD = data; [n] = address;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	sz	0	0	1	0	0	0	0	0	1	Rs				0	Rt2				Rn				Rt							
L										o0																					

### 32-bit (sz == 0)

```
STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release
integer elsize = 32 << (Rs); // ignored by all loads and store-release
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2; integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXP](#).

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.

- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (s == t2) then
  if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
      Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
      assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_NONE rt_unknown = FALSE; // store original value
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
      when Constraint_NONE rn_unknown = FALSE; // address is original base
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

  if n == 31 then
    CheckSPAlignment();
    address = SP[];
  elsif rn_unknown then
    address = bits(64) UNKNOWN;
  else
    address = X[n];

  if rt_unknown then
    data = bits(datasize) UNKNOWN;
  else
    bits(datasize DIV 2) el1 = case memop of
      when MemOp_STORE
        if rt_unknown then
          data = bits(datasize) UNKNOWN;
        elsif pair then
          bits(datasize DIV 2) el1 = X[t];
          bits(datasize DIV 2) el2 = X[t2];
          data = if BigEndian() then el1:el2 else el2:el1;
    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if() then el1 : el2 else el2 : el1;
    else
      data = X[t];

    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
      // This atomic write will be rejected if it does not refer
      // to the same physical locations after address translation.
      Mem[address, dbytes, [address, dbytes, acctype]] = data;
      status = AccType.ATOMIC.ExclusiveMonitorsStatus = data;
      status = (); [s] = ZeroExtend(status, 32);

```

```

when MemOp_LOAD
    // Tell the Exclusive Monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusive Monitor will only be set if all the reads are from the
    // same dbytes-aligned physical address, to allow for the possibility of
    // an atomicity break if the translation is changed between reads.
    AArch64.SetExclusiveMonitors(address, dbytes);

    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acetype];
            if BigEndian() then
                X[t] = data<datasize-1:elsize>;
                X[t2] = data<elsize-1:0>;
            else
                X[t] = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acetype, iswrite, secondstage));
                X[t] = Mem[address + 0, 8, acetype];
                X[t2] = Mem[address + 8, 8, acetype];
            else
                data = MemExclusiveMonitorsStatusX();[address, dbytes, acetype];
X[s] = {t} = ZeroExtend(status, 32);(data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

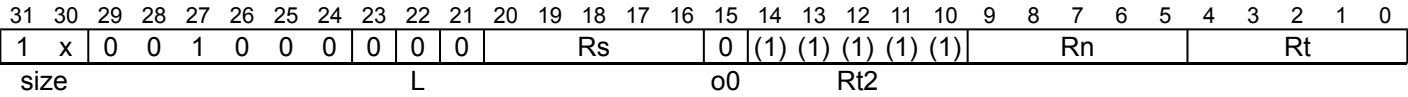
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>



# STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).



## 32-bit (size == 10)

STXR <Ws>, <Wt>, [<Xn|SP>{, #0}]

## 64-bit (size == 11)

STXR <Ws>, <Xt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release
integer elsize = 8 << (Rt2); // ignored by load/store single register
integer s = UInt(size); (Rs); // ignored by all loads and store-release AccType acctype = if o0 == '1'
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXR](#).

## Assembler Symbols

- <Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then if memop ==
  MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
      when Constraint_NONE rt_unknown = FALSE; // store original value
      when Constraint_UNDEF UnallocatedEncoding();
      when Constraint_NOP EndOfInstruction();

if s == n && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
    when Constraint_NONE rn_unknown = FALSE; // address is original base
    when Constraint_UNDEF UnallocatedEncoding();
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elseif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

if rt_unknown then
  data = bits(elsize) UNKNOWN;
else
  data = case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      elsif pair then
        bits(datasize DIV 2) el1 = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if bits(datasize DIV 2) el2 = X[t2];
  data = if BigEndian() then el1 : el2 else el2 : el1;
else
  data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
  // This atomic write will be rejected if it does not refer
  // to the same physical locations after address translation.
  Mem[address, dbytes, [address, dbytes, acctype] = data;

```

```

        status = AccType_ATOMICExclusiveMonitorsStatus] = data;
    status = (); [s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acetype];
            if BigEndian() then
                X[t] = data<datasize-1:elsize>;
                X[t2] = data<elsize-1:0>;
            else
                X[t] = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acetype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acetype];
                    X[t2] = Mem[address + 8, 8, acetype];
                else
                    data = MemExclusiveMonitorsStatusX() [address, dbytes, acetype];
X[s] = [t] = ZeroExtend(status, 32); (data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0					Rs	0	(1)	(1)	(1)	(1)	(1)										
size								L				o0				Rt2						Rn						Rt			

### No offset

STXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release(Rt2); // ignored by load
integer s = UInt(Rs); // ignored by all loads and store-release
AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXRB*.

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NONE rn_unknown = FALSE; // address is original base
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1 : el2 else el2 : el1;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation. (address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.

```

```

Mem[address, 1, [address, dbytes, acctype] = data;
status = AccType ATOMICExclusiveMonitorsStatus] = data;
status = (); [s] = ZeroExtend(status, 32);

when MemOp_LOAD
    // Tell the Exclusive Monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusive Monitor will only be set if all the reads are from the
    // same dbytes-aligned physical address, to allow for the possibility of
    // an atomicity break if the translation is changed between reads.
    AArch64.SetExclusiveMonitors(address, dbytes);

    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t] = bits(datasize) UNKNOWN;
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian() then
                X[t] = data<datasize-1:elsize>;
                X[t2] = data<elsize-1:0>;
            else
                X[t] = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t] = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
            else
                data = MemExclusiveMonitorsStatusX(); [address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32); (data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

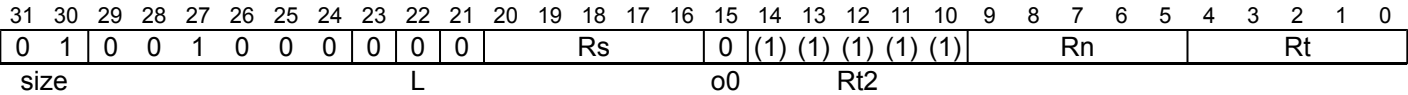
<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>



# STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic.

For information about memory accesses see [Load/Store addressing modes](#).



## No offset

```
STXRH <Ws>, <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release(Rt2); // ignored by load
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler Symbols

- <Ws>

0

1

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

If the operation updates memory.

If the operation fails to update memory.
- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then if memop ==
    MemOp_LOAD && pair && t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UnallocatedEncoding();
            when Constraint_NOP EndOfInstruction();

if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NONE rn_unknown = FALSE; // address is original base
        when Constraint_UNDEF UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1 : el2 else el2 : el1;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation. (address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.

```

```

Mem[address, 2, [address, dbytes, acctype] = data;
status = AccType ATOMICExclusiveMonitorsStatus = data;
status = (); [s] = ZeroExtend(status, 32);

when MemOp_LOAD
// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemExclusiveMonitorsStatusX(); [address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32); (data, regsize);

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	0	1	0	1	1	0	0	1	Rm				option			imm3			Rn				Rd						
op S																															

### 32-bit (sf == 0)

SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

### 64-bit (sf == 1)

SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

## Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '1111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzev;
bit carry_in;

operand2 =if sub_op then
    operand2 = NOT(operand2);
(result, -) = carry_in = '1';
else
    carry_in = '0';

(result, nzev) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

if d == 31 thenif setflags then
    PSTATE.<N,Z,C,V> = nzev;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		1	0	1	0	0	0	1	shift		imm12												Rn				Rd				
op S																															

### 32-bit (sf == 0)

SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}

### 64-bit (sf == 1)

SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12:-(imm12-Zeros(12)), datasize);
  when '1x' ReservedValue();
```

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "shift":

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
bits(datasize) operand2 = imm;
bits(4) nzcvc;
bit carry_in;

operand2 =if sub_op then
  operand2 = NOT(imm);
(result, -) =(operand2);
  carry_in = '1';
else
  carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

if d == 31 thenif setflags then
  PSTATE.<N,Z,C,V> = nzcvc;

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)  
[ISA v82A A64 xml 00bet3.1 OPT](#)



## SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias [NEG \(shifted register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		1	0	0	1	0	1	1	shift		0	Rm				imm6				Rn				Rd							
op S																															

### 32-bit (sf == 0)

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
----------	--

## Alias Conditions

Alias	Is preferred when
<a href="#">NEG (shifted register)</a>	Rn == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzev;
bit carry_in;

operand2 =if sub_op then
    operand2 = NOT(operand2);
(result, -) = carry_in = '1';
else
    carry_in = '0';

(result, nzev) = AddWithCarry(operand1, operand2, '1');(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzev;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias [CMP \(extended register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	0	1	0	1	1	0	0	1	Rm				option			imm3			Rn				Rd						
op S																															

### 32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<R>	Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m>	Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
<extend>	For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '1111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '1111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Alias Conditions

Alias	Is preferred when
<a href="#">CMP (extended register)</a>	Rd == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcvc;
bit carry_in;

operand2 =if sub_op then
operand2 = NOT(operand2);
carry_in = '1';
else
carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcvc;if setflags then
PSTATE.<N,Z,C,V> = nzcvc;

if d == 31 && !setflags then

SP[] = result;
else
X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [CMP \(immediate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		1	1	1	0	0	0	1	shift		imm12										Rn				Rd						
op		S																													

### 32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

### 64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12:~imm12,Zeros(12), datasize);
  when '1x' ReservedValue();
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "shift":

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

## Alias Conditions

Alias	Is preferred when
<a href="#">CMP (immediate)</a>	Rd == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
bits(datasize) operand2 = imm;
bits(4) nzcvc;
bit carry_in;

operand2 =if sub_op then
operand2 = NOT(imm);
(operand2);
carry_in = '1';
else
carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcvc; if setflags then
PSTATE.<N,Z,C,V> = nzcvc;

if d == 31 && !setflags then

SP[] = result;
else
X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#"><u>ISA v82A A64 xml 00bet3.1</u></a>	htmldiff from-	<a href="#"><u>(new)</u></a>
<a href="#"><u>(old)</u></a>	ISA_v82A_A64_xml_00bet3.1	<a href="#"><u>ISA v82A A64 xml 00bet3.1 OPT</u></a>

## SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases [CMP \(shifted register\)](#), and [NEGS](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		1	1	0	1	0	1	1	shift		0	Rm				imm6						Rn						Rd			
op S																															

### 32-bit (sf == 0)

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
	For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">CMP (shifted register)</a>	Rd == '11111'
<a href="#">NEGS</a>	Rn == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcvc;
bit carry_in;

operand2 =if sub_op then
  operand2 = NOT(operand2);
  carry_in = '1';
else
  carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcvc;if setflags then
  PSTATE.<N,Z,C,V> = nzcvc;

X[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



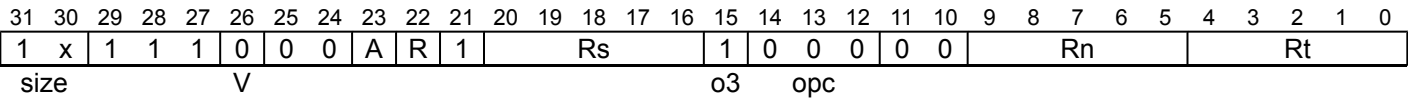
# SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWPL and SWPAL store to memory with release semantics.
- SWP has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).  
 For information about memory accesses see [Load/Store addressing modes](#).

## Integer (ARMv8.1)



### 32-bit, acquire (size == 10 && A == 1 && R == 0)

SWPA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit, acquire and release (size == 10 && A == 1 && R == 1)

SWPAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit, no memory ordering (size == 10 && A == 0 && R == 0)

SWP <Ws>, <Wt>, [<Xn|SP>]

### 32-bit, release (size == 10 && A == 0 && R == 1)

SWPL <Ws>, <Wt>, [<Xn|SP>]

### 64-bit, acquire (size == 11 && A == 1 && R == 0)

SWPA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit, acquire and release (size == 11 && A == 1 && R == 1)

SWPAL <Xs>, <Xt>, [<Xn|SP>]

### 64-bit, no memory ordering (size == 11 && A == 0 && R == 0)

SWP <Xs>, <Xt>, [<Xn|SP>]

### 64-bit, release (size == 11 && A == 0 && R == 1)

SWPL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDRW else AccType ATOMICRW; MemAtomicOp op;
case o3:opc of
when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

// All observers in the shareability domain observe the
// following load and store atomically.case op of
when
MemAtomicOp\_ADD result = data + value;
when MemAtomicOp\_BIC result = data AND NOT(value);
when MemAtomicOp\_EOR result = data EOR value;
when MemAtomicOp\_ORR result = data OR value;
when MemAtomicOp\_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp\_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp\_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp\_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp\_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = value; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	0	1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn					Rt							
size					V										o3					opc														

#### Acquire (A == 1 && R == 0)

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0)

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1)

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

// All observers in the shareability domain observe the
// following load and store atomically.
when
    MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = value; [address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32); (data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u><a href="#">ISA v82A A64 xml 00bet3.1</a></u>	htmldiff from-	<u><a href="#">(new)</a></u>
<u><a href="#">(old)</a></u>	ISA_v82A_A64_xml_00bet3.1	<u><a href="#">ISA v82A A64 xml 00bet3.1 OPT</a></u>

## SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn					Rt				
size					V										o3					opc											

#### Acquire (A == 1 && R == 0)

```
SWPAH <Ws>, <Wt>, [<Xn|SP>]
```

#### Acquire and release (A == 1 && R == 1)

```
SWPALH <Ws>, <Wt>, [<Xn|SP>]
```

#### No memory ordering (A == 0 && R == 0)

```
SWPH <Ws>, <Wt>, [<Xn|SP>]
```

#### Release (A == 0 && R == 1)

```
SWPLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); {Rs};
integer datasize = 8 <<
    UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding; ();
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];

// All observers in the shareability domain observe the
// following load and store atomically.case op of
    when
MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = value;[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

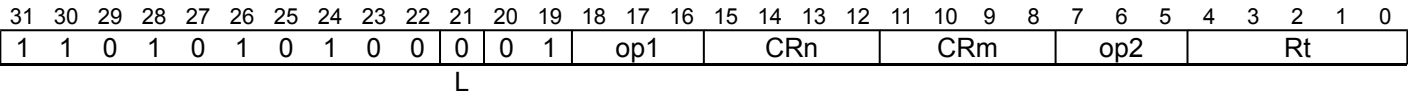
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

# SYS

System instruction. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

This instruction is used by the aliases [AT](#), [DC](#), [IC](#), and [TLBI](#).



## System

SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);
```

```
integer t = UInt(Rt);
```

```
integer sys_op0 = 1;
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm); {CRm};
boolean has_result = (L == '1');
```

## Assembler Symbols

- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">AT</a>	CRn == '0111' && CRm == '100x' && <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_AT</a>
<a href="#">DC</a>	CRn == '0111' && <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_DC</a>
<a href="#">IC</a>	CRn == '0111' && <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_IC</a>
<a href="#">TLBI</a>	CRn == '1000' && <a href="#">SysOp</a> (op1, '1000', CRm, op2) == <a href="#">Sys_TLBI</a>

## Operation

```
if has_result then
    X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, {sys_op0, sys_op1, sys_crn, sys_crm, sys_op2});
```

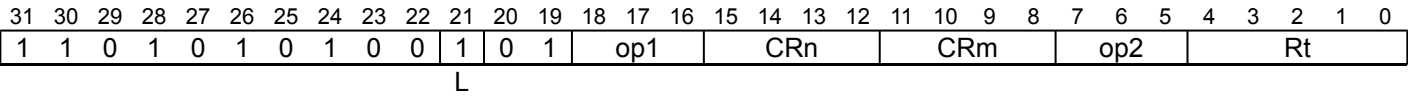
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.



# SYSL

System instruction with result. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.



## System

SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>

```

AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 1;
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm); {CRm};
boolean has_result = (L == '1');
```

## Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

## Operation

```

if has_result then
  X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
  AArch64.SysInstr(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X(1, sys_op1, sys_crn, sys_crm, sys_op2));
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## TBL

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	Rm				0	len	0	0	0	Rn				Rd							
op																															

### Two register table (len == 01)

TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>

### Three register table (len == 10)

TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>

### Four register table (len == 11)

TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>

### Single register table (len == 00)

TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// Create table from registers
for i = 0 to regs-1
for i = 0 to regs - 1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

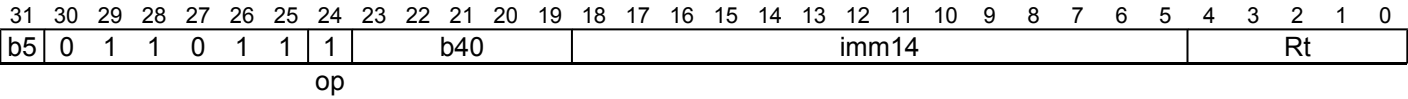
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

# TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



## 14-bit signed PC-relative branch offset

```
TBNZ <R><t>, #<imm>, <label>

integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bit_val = op;
bits(64) offset = SignExtend(imm14:'00', 64);
```

## Assembler Symbols

- <R> Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.
- <t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.
- <imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".
- <label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

## Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == op then if operand<bit_pos> == bit_val then
    BranchTo(PC[] + offset, BranchType_JMP);
```

## TBX

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	Rm				0	len	1	0	0	Rn				Rd							
op																															

### Two register table (len == 01)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>

### Three register table (len == 10)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>

### Four register table (len == 11)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>

### Single register table (len == 00)

TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// Create table from registers
for i = 0 to regs-1
for i = 0 to regs - 1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

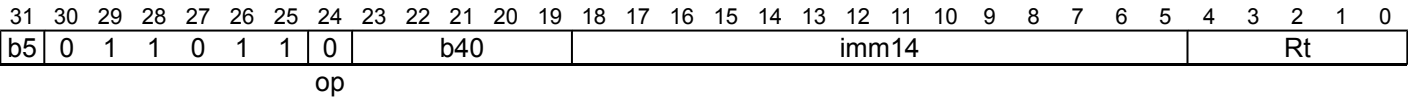
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

# TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



## 14-bit signed PC-relative branch offset

```
TBZ <R><t>, #<imm>, <label>
```

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bit bit_val = op;
bits(64) offset = SignExtend(imm14:'00', 64);
```

## Assembler Symbols

- <R>**

Is a width specifier, encoded in “b5”:
- | b5 | <R> |
|----|-----|
| 0  | W   |
| 1  | X   |
- In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.
- <t>**

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.
- <imm>**

Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".
- <label>**

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

## Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == op thenif operand<bit_pos> == bit_val then
  BranchTo(PC[] + offset, BranchType_JMP);
```

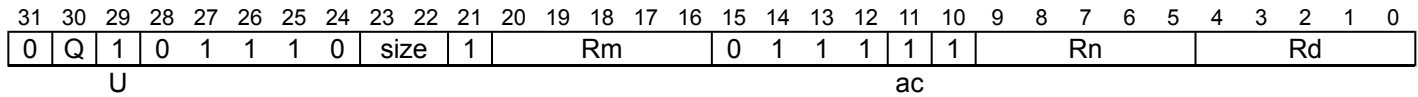
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>; (element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```



[ISA v82A A64 xml 00bet3.1](#)      `htmldiff from-`      [\(new\)](#)  
[\(old\)](#)      `ISA_v82A_A64_xml_00bet3.1`      [ISA v82A A64 xml 00bet3.1 OPT](#)

## UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register, while the UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	1	0	1	1	1	0	size	1	Rm						0	1	0	1	0	0	Rn						Rd					
U										op																							

### Three registers, not all the same type

UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>; (element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

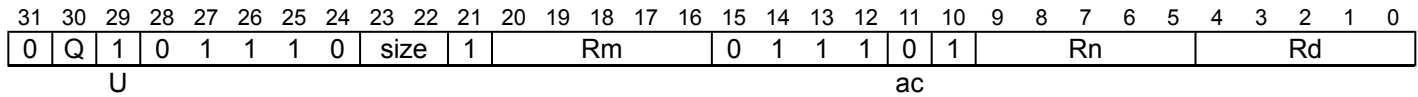
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>; (element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#) [html diff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## UABDL, UABDL2

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABDL instruction extracts each source vector from the lower half of each source register, while the UABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1				Rm			0	1	1	1	0	0				Rn				Rd		
U										op																					

### Three registers, not all the same type

```
UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>; (element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

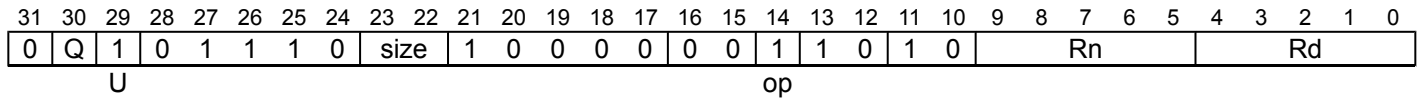
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## UADALP

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Vector

UADALP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>; sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

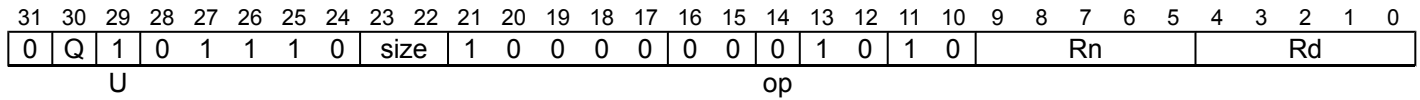
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u><a href="#">ISA v82A A64 xml 00bet3.1</a></u>	htmldiff from-	<u><a href="#">(new)</a></u>
<u><a href="#">(old)</a></u>	ISA_v82A_A64_xml_00bet3.1	<u><a href="#">ISA v82A A64 xml 00bet3.1 OPT</a></u>

## UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Vector

UADDLP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>; sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u><a href="#">ISA v82A A64 xml 00bet3.1</a></u>	htmldiff from-	<u><a href="#">(new)</a></u>
<u><a href="#">(old)</a></u>	ISA_v82A_A64_xml_00bet3.1	<u><a href="#">ISA v82A A64 xml 00bet3.1 OPT</a></u>

## UBFM

Unsigned Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, with zeros in the upper and lower bits.

This instruction is used by the aliases [LSL \(immediate\)](#), [LSR \(immediate\)](#), [UBFIZ](#), [UBFX](#), [UXTB](#), and [UXTH](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		1		0		1		0		0		1		1		0		N		immr						imms						Rn						Rd			
opc																																									

### 32-bit (sf == 0 && N == 0)

UBFM <Wd>, <Wn>, #<immr>, #<imms>

### 64-bit (sf == 1 && N == 1)

UBFM <Xd>, <Xn>, #<immr>, #<imms>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then ease-ope-of
  when '00' inzero = TRUE; extend = TRUE; // SBFM
  when '01' inzero = FALSE; extend = FALSE; // BFM
  when '10' inzero = TRUE; extend = FALSE; // UBFM
  when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt();
R = {immr};
S = UInt(immr);
{imms};
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">LSL (immediate)</a>	32-bit	<code>imms != '011111' &amp;&amp; imms + 1 == immr</code>
<a href="#">LSL (immediate)</a>	64-bit	<code>imms != '111111' &amp;&amp; imms + 1 == immr</code>
<a href="#">LSR (immediate)</a>	32-bit	<code>imms == '011111'</code>
<a href="#">LSR (immediate)</a>	64-bit	<code>imms == '111111'</code>
<a href="#">UBFIZ</a>		<code>UInt(imms) &lt; UInt(immr)</code>
<a href="#">UBFX</a>		<code>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</code>
<a href="#">UXTB</a>		<code>immr == '000000' &amp;&amp; imms == '000111'</code>
<a href="#">UXTH</a>		<code>immr == '000000' &amp;&amp; imms == '001111'</code>

Operation

```
bits(datasize) src =bits(datasize) dst = if inzero then Zeros() else X[n];
// perform bitfield move on low bits
bits(datasize) bot ={d};
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask;
(src, R) AND wmask);

// combine extension bits and result bits// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then
Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT[d] = bot AND tmask; (tmask)) OR (bot AND tmask);
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from-

[\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	0	0	0	0	1	1	scale				Rn				Rd								
rmode										opcode																					

### 32-bit to half-precision (sf == 0 && type == 11) (ARMv8.2)

UCVTF <Hd>, <Wn>, #<fbits>

### 32-bit to single-precision (sf == 0 && type == 00)

UCVTF <Sd>, <Wn>, #<fbits>

### 32-bit to double-precision (sf == 0 && type == 01)

UCVTF <Dd>, <Wn>, #<fbits>

### 64-bit to half-precision (sf == 1 && type == 11) (ARMv8.2)

UCVTF <Hd>, <Xn>, #<fbits>

### 64-bit to single-precision (sf == 1 && type == 00)

UCVTF <Sd>, <Xn>, #<fbits>

### 64-bit to double-precision (sf == 1 && type == 01)

UCVTF <Dd>, <Xn>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPCnvOp op;
FPRounding rounding;
boolean unsigned;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

rounding = case opcode<2:1>:rmode of
  when '00 11' // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPCnvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPCnvOp_CVT_ItoF;
  otherwise
    UnallocatedEncoding(FPCR); {};
```

## Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".  For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
fltval = FixedToFP(intval, fracbits, TRUE, FPCR, rounding);
V[d] = fltval;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>



### UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	0	0	1	1	1	1	0	type	1	0	0	0	1	1	0	0	0	0	0	0	Rn						Rd					
										rmode		opcode																				

**32-bit to half-precision (sf == 0 && type == 11)**  
(ARMv8.2)

UCVTFF <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && type == 00)**

UCVTF <Sd>, <Wn>

**32-bit to double-precision (sf == 0 && type == 01)**

UCVTF <Dd>, <Wn>

**64-bit to half-precision (sf == 1 && type == 11)**  
(ARMv8.2)

UCVTF <Hd>, <Xn>

**64-bit to single-precision (sf == 1 && type == 00)**

UCVTF <Sd>, <Xn>

**64-bit to double-precision (sf == 1 && type == 01)**

UCVTF <Dd>, <Xn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
  when '01'
    fltsize = 64;
  when '10' when '10'
    if opcode<2:1>:rmode != '11-01' then
      UnallocatedEncoding();
  when '11'
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UnallocatedEncoding();

rounding = case opcode<2:1>:rmode of
  when '00-xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01-00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10-00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11-00' // FMOV
    if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11-01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  otherwise
    UnallocatedEncoding(FPCR); ();

```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[n];
  fltval = [d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, TRUE, FPCR, rounding);
V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d] = fltval; [d,part] = fltval;
```

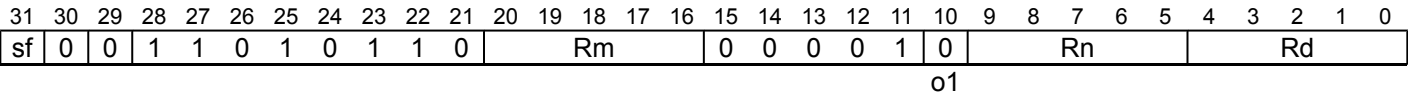
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

# UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.



32-bit (sf == 0)

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
UDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (o1 == '0');
```

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
  result = 0;
else
  result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real((operand1, unsigned)) / Real(Int(operand2, TRUE)));

X[d] = result<datasize-1:0>;
```

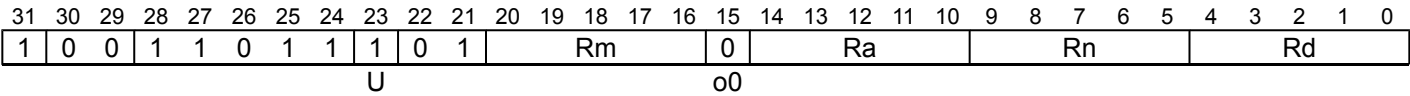
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [UMULL](#).



64-bit

```
UMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Xd>Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm>Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa>Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">UMULL</a>	Ra == '11111'

Operation

```
bits(32) operand1 =bits(datasize) operand1 = X[n];
bits(32) operand2 =bits(datasize) operand2 = X[m];
bits(64) operand3 =bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
    result = Int(operand3, TRUE) + ((operand3, unsigned) - (Int(operand1, TRUE) * (operand1, unsigned) *
else
    result =
Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA\\_v82A\\_A64\\_xml\\_00bet3.1 OPT](#)

## UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M			Rm		0	0	1	0	H	0				Rn					Rd		
U										o2																					

### Vector

UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:



size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)      [html diff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

## UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	1	0	1	1	1	0	size	1	Rm						1	0	0	0	0	0	Rn						Rd							
U										o1																									

### Three registers, not all the same type

UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
[\(old\)](#)

htmldiff from- [\(new\)](#)  
ISA\_v82A\_A64\_xml\_00bet3.1 [ISA v82A A64 xml 00bet3.1 OPT](#)

## UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLSL instruction extracts vector elements from the lower half of the first source register, while the UMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_ELI](#), [CPTL\\_EL2](#), and [CPTL\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M			Rm		0	1	1	0	H	0				Rn					Rd		
U										o2																					

### Vector

UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

ISA v82A A64 xml 00bet3.1 (old)      htldiff from-      (new)  
ISA\_v82A\_A64\_xml\_00bet3.1      ISA v82A A64 xml 00bet3.1 OPT

## UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMLSL instruction extracts each source vector from the lower half of each source register, while the UMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR\_ELI*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	1	0	1	1	1	0	size	1	Rm						1	0	1	0	0	0	Rn						Rd							
U										o1																									

### Three registers, not all the same type

UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#)  
(old)

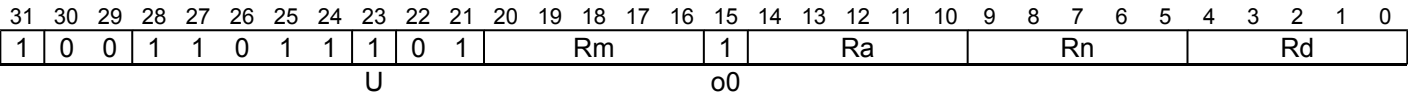
htmldiff from-  
ISA\_v82A\_A64\_xml\_00bet3.1

(new)  
[ISA v82A A64 xml 00bet3.1 OPT](#)

# UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [UMNEGL](#).



## 64-bit

```
UMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">UMNEGL</a>	Ra == '11111'

## Operation

```
bits(32) operand1 =bits(datasize) operand1 = X[n];
bits(32) operand2 =bits(datasize) operand2 = X[m];
bits(64) operand3 =bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
  result = Int(operand3, TRUE) - ((operand3, unsigned) - (Int(operand1, TRUE) * (operand1, unsigned))
else
  result =
Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

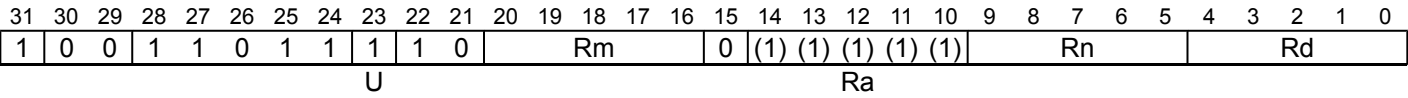
Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.



# UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



## 64-bit

```
UMULH <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra); // ignored by UMULH/SMULH
integer destsize = 64;
integer datasize = destsize;
boolean unsigned = (U == '1');
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

```
bits(64) operand1 =bits(datasize) operand1 = X[n];
bits(64) operand2 =bits(datasize) operand2 = X[m];

integer result;

result = Int(operand1, TRUE) *(operand1, unsigned) * Int(operand2, TRUE);(operand2, unsigned);
X[d] = result<127:64>;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

## UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMULL instruction extracts vector elements from the lower half of the first source register, while the UMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M			Rm			1	0	1	0	H	0			Rn					Rd		
U																															

### Vector

UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>; product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

[ISA v82A A64 xml 00bet3.1](#) [html diff from-](#) [\(new\)](#)  
[\(old\)](#) [ISA\\_v82A\\_A64\\_xml\\_00bet3.1](#) [ISA v82A A64 xml 00bet3.1 OPT](#)

## UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register, while the UMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR\_ELI, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	1	0	1	1	1	0	size	1	Rm						1	1	0	0	0	0	Rn						Rd					
U																																	

### Three registers, not all the same type

UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;[result, e, 2*esize] = (element1 * elem
V[d] = result;
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<a href="#">ISA v82A A64 xml 00bet3.1</a>	htmldiff from-	<a href="#">(new)</a>
<a href="#">(old)</a>	ISA_v82A_A64_xml_00bet3.1	<a href="#">ISA v82A A64 xml 00bet3.1 OPT</a>

## URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see [UHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	1	0	1	1	1	0	size	1	Rm						0	0	0	1	0	1	Rn						Rd				

U

### Three registers of the same type

```
URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>; [result, e, esize] = (element1 + element2 +
V[d] = result;
```

[ISA v82A A64 xml 00bet3.1](#)      [htmldiff from-](#)      [\(new\)](#)  
[\(old\)](#)      ISA\_v82A\_A64\_xml\_00bet3.1      [ISA v82A A64 xml 00bet3.1 OPT](#)

## WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see [Wait For Event mechanism and Send event](#).

As described in [Wait For Event mechanism and Send event](#), the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- [Traps to EL1 of EL0 execution of WFE and WFI instructions](#).
- [Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions](#).
- [Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	1	1	1	1
																CRm				op2											

## System

WFE

```
// Empty.SystemHintOp_op;
case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0010 000'
    op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;
  when '0010 001'
    op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;
  otherwise op = SystemHintOp_NOP;
```



## Operation

```
if case op of
  when SystemHintOp_YIELD Hint_Yield();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
else
  if PSTATE.EL == EL0 then
    // Check for traps described by the OS which may be EL1 or EL2.
    AArch64.CheckForWFXTrap(EL1, TRUE);
  if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
    // Check for traps described by the Hypervisor.
    AArch64.CheckForWFXTrap(EL2, TRUE);
  if HaveEL(EL3) && PSTATE.EL != EL3 then
    // Check for traps described by the Secure Monitor.
    AArch64.CheckForWFXTrap(EL3, TRUE);
  WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, FALSE);
        if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
          // Check for traps described by the Hypervisor.
          AArch64.CheckForWFXTrap(EL2, FALSE);
          if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFXTrap(EL3, FALSE);
          WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBRegDomain_FullSystem, MBRegTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

otherwise // do nothing
```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see [Wait For Interrupt](#).

As described in [Wait For Interrupt](#), the execution of a WFI instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- [Traps to EL1 of EL0 execution of WFE and WFI instructions](#).
- [Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions](#).
- [Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1
																CRm				op2											

System

WFI

```
// Empty.SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0010 000'
    op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;
  when '0010 001'
    op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;
  otherwise op = SystemHintOp_NOP;
```

## Operation

```

if !case op of
  when SystemHintOp_YIELD Hint_Yield();

  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, TRUE);
        if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
          // Check for traps described by the Hypervisor.
          AArch64.CheckForWFXTrap(EL2, TRUE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
          // Check for traps described by the Secure Monitor.
          AArch64.CheckForWFXTrap(EL3, TRUE);
        WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, FALSE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, FALSE);
      WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBRegDomain_FullSystem, MBRegTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

otherwise // do nothing

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>

## YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see [The YIELD instruction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
																CRm				op2											

## System

YIELD

```
// Empty.SystemHintOp op;
case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0010 000'
    op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;
  when '0010 001'
    op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;
  otherwise op = SystemHintOp_NOP;
```

## Operation

```

case op of
  when SystemHintOp_YIELD
    Hint_Yield();

  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, TRUE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, TRUE);
      WaitForEvent();

  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, FALSE);
      if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, FALSE);
      WaitForInterrupt();

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBRegDomain_FullSystem, MBRegTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();();

  otherwise // do nothing

```

Internal version only: isa v25.07, AdvSIMD v23.0, pseudocode v31.3

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Confidential.

<u>ISA v82A A64 xml 00bet3.1</u>	htmldiff from-	<u>(new)</u>
<u>(old)</u>	ISA_v82A_A64_xml_00bet3.1	<u>ISA v82A A64 xml 00bet3.1 OPT</u>