

# Arm® Instruction Set

**Version 1.0**

**Reference Guide**

**arm**

# Arm® Instruction Set

## Reference Guide

Copyright © 2018 Arm Limited or its affiliates. All rights reserved.

### Release Information

### Document History

Issue	Date	Confidentiality	Change
0100-00	25 October 2018	Non-Confidential	First Release

### Non-Confidential 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. **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 third party 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 click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2018 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

**Confidentiality Status**

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

**Product Status**

The information in this document is Final, that is for a developed product.

**Web Address**

<http://www.arm.com>



# Contents

# Arm® Instruction Set Reference Guide

## Preface

<i>About this book</i> .....	42
------------------------------	----

## Part A

## Instruction Set Overview

### Chapter A1

#### Overview of the Arm® Architecture

A1.1 <i>About the Arm® architecture</i> .....	A1-48
A1.2 <i>Differences between the A64, A32, and T32 instruction sets</i> .....	A1-49
A1.3 <i>Changing between AArch64 and AArch32 states</i> .....	A1-50
A1.4 <i>Advanced SIMD</i> .....	A1-51
A1.5 <i>Floating-point hardware</i> .....	A1-52

### Chapter A2

#### Overview of AArch32 state

A2.1 <i>Changing between A32 and T32 instruction set states</i> .....	A2-54
A2.2 <i>Processor modes, and privileged and unprivileged software execution</i> .....	A2-55
A2.3 <i>Processor modes in Armv6-M, Armv7-M, and Armv8-M</i> .....	A2-56
A2.4 <i>Registers in AArch32 state</i> .....	A2-57
A2.5 <i>General-purpose registers in AArch32 state</i> .....	A2-59
A2.6 <i>Register accesses in AArch32 state</i> .....	A2-60
A2.7 <i>Predeclared core register names in AArch32 state</i> .....	A2-61
A2.8 <i>Predeclared extension register names in AArch32 state</i> .....	A2-62
A2.9 <i>Program Counter in AArch32 state</i> .....	A2-63
A2.10 <i>The Q flag in AArch32 state</i> .....	A2-64

A2.11	<i>Application Program Status Register</i> .....	A2-65
A2.12	<i>Current Program Status Register in AArch32 state</i> .....	A2-66
A2.13	<i>Saved Program Status Registers in AArch32 state</i> .....	A2-67
A2.14	<i>A32 and T32 instruction set overview</i> .....	A2-68
A2.15	<i>Access to the inline barrel shifter in AArch32 state</i> .....	A2-69

## Chapter A3

### Overview of AArch64 state

A3.1	<i>Registers in AArch64 state</i> .....	A3-72
A3.2	<i>Exception levels</i> .....	A3-73
A3.3	<i>Link registers</i> .....	A3-74
A3.4	<i>Stack Pointer register</i> .....	A3-75
A3.5	<i>Predeclared core register names in AArch64 state</i> .....	A3-76
A3.6	<i>Predeclared extension register names in AArch64 state</i> .....	A3-77
A3.7	<i>Program Counter in AArch64 state</i> .....	A3-78
A3.8	<i>Conditional execution in AArch64 state</i> .....	A3-79
A3.9	<i>The Q flag in AArch64 state</i> .....	A3-80
A3.10	<i>Process State</i> .....	A3-81
A3.11	<i>Saved Program Status Registers in AArch64 state</i> .....	A3-82
A3.12	<i>A64 instruction set overview</i> .....	A3-83

## Part B

### Advanced SIMD and Floating-point Programming

## Chapter B1

### Advanced SIMD Programming

B1.1	<i>Architecture support for Advanced SIMD</i> .....	B1-88
B1.2	<i>Extension register bank mapping for Advanced SIMD in AArch32 state</i> .....	B1-89
B1.3	<i>Extension register bank mapping for Advanced SIMD in AArch64 state</i> .....	B1-91
B1.4	<i>Views of the Advanced SIMD register bank in AArch32 state</i> .....	B1-93
B1.5	<i>Views of the Advanced SIMD register bank in AArch64 state</i> .....	B1-94
B1.6	<i>Differences between A32/T32 and A64 Advanced SIMD instruction syntax</i> .....	B1-95
B1.7	<i>Load values to Advanced SIMD registers</i> .....	B1-97
B1.8	<i>Conditional execution of A32/T32 Advanced SIMD instructions</i> .....	B1-98
B1.9	<i>Floating-point exceptions for Advanced SIMD in A32/T32 instructions</i> .....	B1-99
B1.10	<i>Advanced SIMD data types in A32/T32 instructions</i> .....	B1-100
B1.11	<i>Polynomial arithmetic over {0, 1}</i> .....	B1-101
B1.12	<i>Advanced SIMD vectors</i> .....	B1-102
B1.13	<i>Normal, long, wide, and narrow Advanced SIMD instructions</i> .....	B1-103
B1.14	<i>Saturating Advanced SIMD instructions</i> .....	B1-104
B1.15	<i>Advanced SIMD scalars</i> .....	B1-105
B1.16	<i>Extended notation extension for Advanced SIMD in A32/T32 code</i> .....	B1-106
B1.17	<i>Advanced SIMD system registers in AArch32 state</i> .....	B1-107
B1.18	<i>Flush-to-zero mode in Advanced SIMD</i> .....	B1-108
B1.19	<i>When to use flush-to-zero mode in Advanced SIMD</i> .....	B1-109
B1.20	<i>The effects of using flush-to-zero mode in Advanced SIMD</i> .....	B1-110
B1.21	<i>Advanced SIMD operations not affected by flush-to-zero mode</i> .....	B1-111

## Chapter B2

### Floating-point Programming

B2.1	<i>Architecture support for floating-point</i> .....	B2-114
B2.2	<i>Extension register bank mapping for floating-point in AArch32 state</i> .....	B2-115
B2.3	<i>Extension register bank mapping in AArch64 state</i> .....	B2-117
B2.4	<i>Views of the floating-point extension register bank in AArch32 state</i> .....	B2-118

B2.5	<i>Views of the floating-point extension register bank in AArch64 state</i>	B2-119
B2.6	<i>Differences between A32/T32 and A64 floating-point instruction syntax</i>	B2-120
B2.7	<i>Load values to floating-point registers</i>	B2-121
B2.8	<i>Conditional execution of A32/T32 floating-point instructions</i>	B2-122
B2.9	<i>Floating-point exceptions for floating-point in A32/T32 instructions</i>	B2-123
B2.10	<i>Floating-point data types in A32/T32 instructions</i>	B2-124
B2.11	<i>Extended notation extension for floating-point in A32/T32 code</i>	B2-125
B2.12	<i>Floating-point system registers in AArch32 state</i>	B2-126
B2.13	<i>Flush-to-zero mode in floating-point</i>	B2-127
B2.14	<i>When to use flush-to-zero mode in floating-point</i>	B2-128
B2.15	<i>The effects of using flush-to-zero mode in floating-point</i>	B2-129
B2.16	<i>Floating-point operations not affected by flush-to-zero mode</i>	B2-130

## Part C

### A32/T32 Instruction Set Reference

#### Chapter C1

##### Condition Codes

C1.1	<i>Conditional instructions</i>	C1-134
C1.2	<i>Conditional execution in A32 code</i>	C1-135
C1.3	<i>Conditional execution in T32 code</i>	C1-136
C1.4	<i>Condition flags</i>	C1-137
C1.5	<i>Updates to the condition flags in A32/T32 code</i>	C1-138
C1.6	<i>Floating-point instructions that update the condition flags</i>	C1-139
C1.7	<i>Carry flag</i>	C1-140
C1.8	<i>Overflow flag</i>	C1-141
C1.9	<i>Condition code suffixes</i>	C1-142
C1.10	<i>Condition code suffixes and related flags</i>	C1-143
C1.11	<i>Comparison of condition code meanings in integer and floating-point code</i>	C1-144
C1.12	<i>Benefits of using conditional execution in A32 and T32 code</i>	C1-146
C1.13	<i>Example showing the benefits of conditional instructions in A32 and T32 code</i>	C1-147
C1.14	<i>Optimization for execution speed</i>	C1-150

#### Chapter C2

##### A32 and T32 Instructions

C2.1	<i>A32 and T32 instruction summary</i>	C2-156
C2.2	<i>Instruction width specifiers</i>	C2-161
C2.3	<i>Flexible second operand (Operand2)</i>	C2-162
C2.4	<i>Syntax of Operand2 as a constant</i>	C2-163
C2.5	<i>Syntax of Operand2 as a register with optional shift</i>	C2-164
C2.6	<i>Shift operations</i>	C2-165
C2.7	<i>Saturating instructions</i>	C2-168
C2.8	<i>ADC</i>	C2-169
C2.9	<i>ADD</i>	C2-171
C2.10	<i>ADR (PC-relative)</i>	C2-174
C2.11	<i>ADR (register-relative)</i>	C2-176
C2.12	<i>AND</i>	C2-178
C2.13	<i>ASR</i>	C2-180
C2.14	<i>B</i>	C2-182
C2.15	<i>BFC</i>	C2-184
C2.16	<i>BFI</i>	C2-185
C2.17	<i>BIC</i>	C2-186
C2.18	<i>BKPT</i>	C2-188

C2.19	<i>BL</i>	C2-189
C2.20	<i>BLX, BLXNS</i>	C2-190
C2.21	<i>BX, BXNS</i>	C2-192
C2.22	<i>BXJ</i>	C2-194
C2.23	<i>CBZ and CBNZ</i>	C2-195
C2.24	<i>CDP and CDP2</i>	C2-196
C2.25	<i>CLREX</i>	C2-197
C2.26	<i>CLZ</i>	C2-198
C2.27	<i>CMP and CMN</i>	C2-199
C2.28	<i>CPS</i>	C2-201
C2.29	<i>CRC32</i>	C2-203
C2.30	<i>CRC32C</i>	C2-204
C2.31	<i>CSDB</i>	C2-205
C2.32	<i>DBG</i>	C2-207
C2.33	<i>DCPS1 (T32 instruction)</i>	C2-208
C2.34	<i>DCPS2 (T32 instruction)</i>	C2-209
C2.35	<i>DCPS3 (T32 instruction)</i>	C2-210
C2.36	<i>DMB</i>	C2-211
C2.37	<i>DSB</i>	C2-213
C2.38	<i>EOR</i>	C2-215
C2.39	<i>ERET</i>	C2-217
C2.40	<i>ESB</i>	C2-218
C2.41	<i>HLT</i>	C2-219
C2.42	<i>HVC</i>	C2-220
C2.43	<i>ISB</i>	C2-221
C2.44	<i>IT</i>	C2-222
C2.45	<i>LDA</i>	C2-225
C2.46	<i>LDAEX</i>	C2-226
C2.47	<i>LDC and LDC2</i>	C2-228
C2.48	<i>LDM</i>	C2-230
C2.49	<i>LDR (immediate offset)</i>	C2-232
C2.50	<i>LDR (PC-relative)</i>	C2-234
C2.51	<i>LDR (register offset)</i>	C2-236
C2.52	<i>LDR (register-relative)</i>	C2-238
C2.53	<i>LDR, unprivileged</i>	C2-240
C2.54	<i>LDREX</i>	C2-242
C2.55	<i>LSL</i>	C2-244
C2.56	<i>LSR</i>	C2-246
C2.57	<i>MCR and MCR2</i>	C2-248
C2.58	<i>MCRR and MCRR2</i>	C2-249
C2.59	<i>MLA</i>	C2-250
C2.60	<i>MLS</i>	C2-251
C2.61	<i>MOV</i>	C2-252
C2.62	<i>MOVT</i>	C2-254
C2.63	<i>MRC and MRC2</i>	C2-255
C2.64	<i>MRRC and MRRC2</i>	C2-256
C2.65	<i>MRS (PSR to general-purpose register)</i>	C2-257
C2.66	<i>MRS (system coprocessor register to general-purpose register)</i>	C2-259
C2.67	<i>MSR (general-purpose register to system coprocessor register)</i>	C2-260
C2.68	<i>MSR (general-purpose register to PSR)</i>	C2-261

C2.69	MUL .....	C2-263
C2.70	MVN .....	C2-264
C2.71	NOP .....	C2-266
C2.72	ORN ( <i>T32 only</i> ) .....	C2-267
C2.73	ORR .....	C2-268
C2.74	PKHBT and PKHTB .....	C2-270
C2.75	PLD, PLDW, and PLI .....	C2-272
C2.76	POP .....	C2-274
C2.77	PUSH .....	C2-275
C2.78	QADD .....	C2-276
C2.79	QADD8 .....	C2-277
C2.80	QADD16 .....	C2-278
C2.81	QASX .....	C2-279
C2.82	QDADD .....	C2-280
C2.83	QDSUB .....	C2-281
C2.84	QSAX .....	C2-282
C2.85	QSUB .....	C2-283
C2.86	QSUB8 .....	C2-284
C2.87	QSUB16 .....	C2-285
C2.88	RBIT .....	C2-286
C2.89	REV .....	C2-287
C2.90	REV16 .....	C2-288
C2.91	REVSH .....	C2-289
C2.92	RFE .....	C2-290
C2.93	ROR .....	C2-292
C2.94	RRX .....	C2-294
C2.95	RSB .....	C2-296
C2.96	RSC .....	C2-298
C2.97	SADD8 .....	C2-300
C2.98	SADD16 .....	C2-302
C2.99	SASX .....	C2-304
C2.100	SBC .....	C2-306
C2.101	SBFX .....	C2-308
C2.102	SDIV .....	C2-309
C2.103	SEL .....	C2-310
C2.104	SETEND .....	C2-312
C2.105	SETPAN .....	C2-313
C2.106	SEV .....	C2-314
C2.107	SEVL .....	C2-315
C2.108	SG .....	C2-316
C2.109	SHADD8 .....	C2-317
C2.110	SHADD16 .....	C2-318
C2.111	SHASX .....	C2-319
C2.112	SHSAX .....	C2-320
C2.113	SHSUB8 .....	C2-321
C2.114	SHSUB16 .....	C2-322
C2.115	SMC .....	C2-323
C2.116	SMLAxy .....	C2-324
C2.117	SMLAD .....	C2-326
C2.118	SMLAL .....	C2-327

C2.119	SMLALD .....	C2-328
C2.120	SMLALxy .....	C2-329
C2.121	SMLAWy .....	C2-331
C2.122	SMLSD .....	C2-332
C2.123	SMLS LD .....	C2-333
C2.124	SMMLA .....	C2-334
C2.125	SMM LS .....	C2-335
C2.126	SMMUL .....	C2-336
C2.127	SMUAD .....	C2-337
C2.128	SMULxy .....	C2-338
C2.129	SMULL .....	C2-339
C2.130	SMULWy .....	C2-340
C2.131	SMUSD .....	C2-341
C2.132	SRS .....	C2-342
C2.133	SSAT .....	C2-344
C2.134	SSAT16 .....	C2-345
C2.135	SSAX .....	C2-346
C2.136	SSUB8 .....	C2-348
C2.137	SSUB16 .....	C2-350
C2.138	STC and STC2 .....	C2-352
C2.139	STL .....	C2-354
C2.140	STLEX .....	C2-355
C2.141	STM .....	C2-357
C2.142	STR ( <i>immediate offset</i> ) .....	C2-359
C2.143	STR ( <i>register offset</i> ) .....	C2-361
C2.144	STR, unprivileged .....	C2-363
C2.145	STREX .....	C2-365
C2.146	SUB .....	C2-367
C2.147	SUBS pc, lr .....	C2-370
C2.148	SVC .....	C2-372
C2.149	SWP and SWPB .....	C2-373
C2.150	SXTAB .....	C2-374
C2.151	SXTAB16 .....	C2-376
C2.152	SXTAH .....	C2-378
C2.153	SXTB .....	C2-380
C2.154	SXTB16 .....	C2-382
C2.155	SXTH .....	C2-383
C2.156	SYS .....	C2-385
C2.157	TBB and TBH .....	C2-386
C2.158	TEQ .....	C2-387
C2.159	TST .....	C2-389
C2.160	TT, TTT, TTA, TTAT .....	C2-391
C2.161	UADD8 .....	C2-393
C2.162	UADD16 .....	C2-395
C2.163	UASX .....	C2-397
C2.164	UBFX .....	C2-399
C2.165	UDF .....	C2-400
C2.166	UDIV .....	C2-401
C2.167	UHADD8 .....	C2-402
C2.168	UHADD16 .....	C2-403

C2.169	UHASX .....	C2-404
C2.170	UHSAX .....	C2-405
C2.171	UHSUB8 .....	C2-406
C2.172	UHSUB16 .....	C2-407
C2.173	UMAAL .....	C2-408
C2.174	UMLAL .....	C2-409
C2.175	UMULL .....	C2-410
C2.176	UQADD8 .....	C2-411
C2.177	UQADD16 .....	C2-412
C2.178	UQASX .....	C2-413
C2.179	UQSAX .....	C2-414
C2.180	UQSUB8 .....	C2-415
C2.181	UQSUB16 .....	C2-416
C2.182	USAD8 .....	C2-417
C2.183	USADA8 .....	C2-418
C2.184	USAT .....	C2-419
C2.185	USAT16 .....	C2-420
C2.186	USAX .....	C2-421
C2.187	USUB8 .....	C2-423
C2.188	USUB16 .....	C2-425
C2.189	UXTAB .....	C2-426
C2.190	UXTAB16 .....	C2-428
C2.191	UXTAH .....	C2-430
C2.192	UXTB .....	C2-432
C2.193	UXTB16 .....	C2-434
C2.194	UXTH .....	C2-435
C2.195	WFE .....	C2-437
C2.196	WFI .....	C2-438
C2.197	YIELD .....	C2-439

## Chapter C3

### Advanced SIMD Instructions (32-bit)

C3.1	Summary of Advanced SIMD instructions .....	C3-445
C3.2	Summary of shared Advanced SIMD and floating-point instructions .....	C3-448
C3.3	Interleaving provided by load and store element and structure instructions .....	C3-449
C3.4	Alignment restrictions in load and store element and structure instructions .....	C3-450
C3.5	FLDMDBX, FLDMIAX .....	C3-451
C3.6	FSTMDBX, FSTMIAX .....	C3-452
C3.7	VABA and VABAL .....	C3-453
C3.8	VABD and VABDL .....	C3-454
C3.9	VABS .....	C3-455
C3.10	VACLE, VACLT, VACGE and VACGT .....	C3-456
C3.11	VADD .....	C3-457
C3.12	VADDHN .....	C3-458
C3.13	VADDL and VADDW .....	C3-459
C3.14	VAND (immediate) .....	C3-460
C3.15	VAND (register) .....	C3-461
C3.16	VBIC (immediate) .....	C3-462
C3.17	VBIC (register) .....	C3-463
C3.18	VBIF .....	C3-464
C3.19	VBIT .....	C3-465

C3.20	VBSL .....	C3-466
C3.21	VCADD .....	C3-467
C3.22	VCEQ ( <i>immediate #0</i> ) .....	C3-468
C3.23	VCEQ ( <i>register</i> ) .....	C3-469
C3.24	VCGE ( <i>immediate #0</i> ) .....	C3-470
C3.25	VCGE ( <i>register</i> ) .....	C3-471
C3.26	VCGT ( <i>immediate #0</i> ) .....	C3-472
C3.27	VCGT ( <i>register</i> ) .....	C3-473
C3.28	VCLE ( <i>immediate #0</i> ) .....	C3-474
C3.29	VCLS .....	C3-475
C3.30	VCLE ( <i>register</i> ) .....	C3-476
C3.31	VCLT ( <i>immediate #0</i> ) .....	C3-477
C3.32	VCLT ( <i>register</i> ) .....	C3-478
C3.33	VCLZ .....	C3-479
C3.34	VCMLA .....	C3-480
C3.35	VCMLA ( <i>by element</i> ) .....	C3-481
C3.36	VCNT .....	C3-482
C3.37	VCVT ( <i>between fixed-point or integer, and floating-point</i> ) .....	C3-483
C3.38	VCVT ( <i>between half-precision and single-precision floating-point</i> ) .....	C3-484
C3.39	VCVT ( <i>from floating-point to integer with directed rounding modes</i> ) .....	C3-485
C3.40	VCVTB, VCVTT ( <i>between half-precision and double-precision</i> ) .....	C3-486
C3.41	VDUP .....	C3-487
C3.42	VEOR .....	C3-488
C3.43	VEXT .....	C3-489
C3.44	VFMA, VFMS .....	C3-490
C3.45	VFMAL ( <i>by scalar</i> ) .....	C3-491
C3.46	VFMAL ( <i>vector</i> ) .....	C3-492
C3.47	VFMSL ( <i>by scalar</i> ) .....	C3-493
C3.48	VFMSL ( <i>vector</i> ) .....	C3-494
C3.49	VHADD .....	C3-495
C3.50	VHSUB .....	C3-496
C3.51	VLDn ( <i>single n-element structure to one lane</i> ) .....	C3-497
C3.52	VLDn ( <i>single n-element structure to all lanes</i> ) .....	C3-499
C3.53	VLDn ( <i>multiple n-element structures</i> ) .....	C3-501
C3.54	VLDM .....	C3-503
C3.55	VLDR .....	C3-504
C3.56	VLDR ( <i>post-increment and pre-decrement</i> ) .....	C3-505
C3.57	VLDR pseudo-instruction .....	C3-506
C3.58	VMAX and VMIN .....	C3-507
C3.59	VMAXNM, VMINNM .....	C3-508
C3.60	VMLA .....	C3-509
C3.61	VMLA ( <i>by scalar</i> ) .....	C3-510
C3.62	VMLAL ( <i>by scalar</i> ) .....	C3-511
C3.63	VMLAL .....	C3-512
C3.64	VMLS ( <i>by scalar</i> ) .....	C3-513
C3.65	VMLS .....	C3-514
C3.66	VMLSL .....	C3-515
C3.67	VMLSL ( <i>by scalar</i> ) .....	C3-516
C3.68	VMOV ( <i>immediate</i> ) .....	C3-517
C3.69	VMOV ( <i>register</i> ) .....	C3-518

C3.70	VMOV (between two general-purpose registers and a 64-bit extension register) ....	C3-519
C3.71	VMOV (between a general-purpose register and an Advanced SIMD scalar) ....	C3-520
C3.72	VMOVL .....	C3-521
C3.73	VMOVN .....	C3-522
C3.74	VMOV2 .....	C3-523
C3.75	VMRS .....	C3-524
C3.76	VMSR .....	C3-525
C3.77	VMUL .....	C3-526
C3.78	VMUL (by scalar) .....	C3-527
C3.79	VMULL .....	C3-528
C3.80	VMULL (by scalar) .....	C3-529
C3.81	VMVN (register) .....	C3-530
C3.82	VMVN (immediate) .....	C3-531
C3.83	VNEG .....	C3-532
C3.84	VORN (register) .....	C3-533
C3.85	VORN (immediate) .....	C3-534
C3.86	VORR (register) .....	C3-535
C3.87	VORR (immediate) .....	C3-536
C3.88	VPADAL .....	C3-537
C3.89	VPADD .....	C3-538
C3.90	VPADDL .....	C3-539
C3.91	VPMAX and VPMIN .....	C3-540
C3.92	VPOP .....	C3-541
C3.93	VPUSH .....	C3-542
C3.94	VQABS .....	C3-543
C3.95	VQADD .....	C3-544
C3.96	VQDMMLA and VQDMMLS (by vector or by scalar) .....	C3-545
C3.97	VQDMULH (by vector or by scalar) .....	C3-546
C3.98	VQDMULL (by vector or by scalar) .....	C3-547
C3.99	VQMOVN and VQMOVUN .....	C3-548
C3.100	VQNEG .....	C3-549
C3.101	VQRDMULH (by vector or by scalar) .....	C3-550
C3.102	VQRSHL (by signed variable) .....	C3-551
C3.103	VQRSHRN and VQRSHRUN (by immediate) .....	C3-552
C3.104	VQSHL (by signed variable) .....	C3-553
C3.105	VQSHL and VQSHLU (by immediate) .....	C3-554
C3.106	VQSHRN and VQSHRUN (by immediate) .....	C3-555
C3.107	VQSUB .....	C3-556
C3.108	VRADDHN .....	C3-557
C3.109	VRECPE .....	C3-558
C3.110	VRECPs .....	C3-559
C3.111	VREV16, VREV32, and VREV64 .....	C3-560
C3.112	VRHADD .....	C3-561
C3.113	VRSHL (by signed variable) .....	C3-562
C3.114	VRSHR (by immediate) .....	C3-563
C3.115	VRSHRN (by immediate) .....	C3-564
C3.116	VRINT .....	C3-565
C3.117	VRSQRTE .....	C3-566
C3.118	VRSQRTS .....	C3-567

C3.119	VRSRA ( <i>by immediate</i> ) .....	C3-568
C3.120	VRSUBHN .....	C3-569
C3.121	VSDOT ( <i>vector</i> ) .....	C3-570
C3.122	VSDOT ( <i>by element</i> ) .....	C3-571
C3.123	VSHL ( <i>by immediate</i> ) .....	C3-572
C3.124	VSHL ( <i>by signed variable</i> ) .....	C3-573
C3.125	VSHLL ( <i>by immediate</i> ) .....	C3-574
C3.126	VSHR ( <i>by immediate</i> ) .....	C3-575
C3.127	VSHRN ( <i>by immediate</i> ) .....	C3-576
C3.128	VSLI .....	C3-577
C3.129	VSRA ( <i>by immediate</i> ) .....	C3-578
C3.130	VSRI .....	C3-579
C3.131	VSTM .....	C3-580
C3.132	VSTn ( <i>multiple n-element structures</i> ) .....	C3-581
C3.133	VSTn ( <i>single n-element structure to one lane</i> ) .....	C3-583
C3.134	VSTR .....	C3-585
C3.135	VSTR ( <i>post-increment and pre-decrement</i> ) .....	C3-586
C3.136	VSUB .....	C3-587
C3.137	VSUBHN .....	C3-588
C3.138	VSUBL and VSUBW .....	C3-589
C3.139	VSWP .....	C3-590
C3.140	VTBL and VTBX .....	C3-591
C3.141	VTRN .....	C3-592
C3.142	VTST .....	C3-593
C3.143	VUDOT ( <i>vector</i> ) .....	C3-594
C3.144	VUDOT ( <i>by element</i> ) .....	C3-595
C3.145	VUZP .....	C3-596
C3.146	VZIP .....	C3-597

## Chapter C4

### Floating-point Instructions (32-bit)

C4.1	Summary of floating-point instructions .....	C4-601
C4.2	VABS ( <i>floating-point</i> ) .....	C4-603
C4.3	VADD ( <i>floating-point</i> ) .....	C4-604
C4.4	VCMP, VCMPE .....	C4-605
C4.5	VCVT ( <i>between single-precision and double-precision</i> ) .....	C4-606
C4.6	VCVT ( <i>between floating-point and integer</i> ) .....	C4-607
C4.7	VCVT ( <i>from floating-point to integer with directed rounding modes</i> ) .....	C4-608
C4.8	VCVT ( <i>between floating-point and fixed-point</i> ) .....	C4-609
C4.9	VCVTB, VCVTT ( <i>half-precision extension</i> ) .....	C4-610
C4.10	VCVTB, VCVTT ( <i>between half-precision and double-precision</i> ) .....	C4-611
C4.11	VDIV .....	C4-612
C4.12	VFMA, VFMS, VFNMA, VFNMS ( <i>floating-point</i> ) .....	C4-613
C4.13	VJCVT .....	C4-614
C4.14	VLDM ( <i>floating-point</i> ) .....	C4-615
C4.15	VLDR ( <i>floating-point</i> ) .....	C4-616
C4.16	VLDR ( <i>post-increment and pre-decrement, floating-point</i> ) .....	C4-617
C4.17	VLLDM .....	C4-618
C4.18	VLSTM .....	C4-619
C4.19	VMAXNM, VMINNM ( <i>floating-point</i> ) .....	C4-620
C4.20	VMLA ( <i>floating-point</i> ) .....	C4-621

C4.21	VMLS ( <i>floating-point</i> ) .....	C4-622
C4.22	VMOV ( <i>floating-point</i> ) .....	C4-623
C4.23	VMOV ( <i>between one general-purpose register and single precision floating-point register</i> ) .....	C4-624
C4.24	VMOV ( <i>between two general-purpose registers and one or two extension registers</i> ) .....	C4-625
C4.25	VMOV ( <i>between a general-purpose register and half a double precision floating-point register</i> ) .....	C4-626
C4.26	VMRS ( <i>floating-point</i> ) .....	C4-627
C4.27	VMSR ( <i>floating-point</i> ) .....	C4-628
C4.28	VMUL ( <i>floating-point</i> ) .....	C4-629
C4.29	VNEG ( <i>floating-point</i> ) .....	C4-630
C4.30	VNMLA ( <i>floating-point</i> ) .....	C4-631
C4.31	VNMLS ( <i>floating-point</i> ) .....	C4-632
C4.32	VNMUL ( <i>floating-point</i> ) .....	C4-633
C4.33	VPOP ( <i>floating-point</i> ) .....	C4-634
C4.34	VPUSH ( <i>floating-point</i> ) .....	C4-635
C4.35	VRINT ( <i>floating-point</i> ) .....	C4-636
C4.36	VSEL .....	C4-637
C4.37	VSQRT .....	C4-638
C4.38	VSTM ( <i>floating-point</i> ) .....	C4-639
C4.39	VSTR ( <i>floating-point</i> ) .....	C4-640
C4.40	VSTR ( <i>post-increment and pre-decrement, floating-point</i> ) .....	C4-641
C4.41	VSUB ( <i>floating-point</i> ) .....	C4-642

## Chapter C5

### A32/T32 Cryptographic Algorithms

C5.1	A32/T32 Cryptographic instructions .....	C5-644
------	------------------------------------------	--------

## Part D

### A64 Instruction Set Reference

#### Chapter D1

##### Condition Codes

D1.1	Conditional execution in A64 code .....	D1-648
D1.2	Condition flags .....	D1-649
D1.3	Updates to the condition flags in A64 code .....	D1-650
D1.4	Floating-point instructions that update the condition flags .....	D1-651
D1.5	Carry flag .....	D1-652
D1.6	Overflow flag .....	D1-653
D1.7	Condition code suffixes .....	D1-654
D1.8	Condition code suffixes and related flags .....	D1-655
D1.9	Optimization for execution speed .....	D1-656

#### Chapter D2

##### A64 General Instructions

D2.1	A64 instructions in alphabetical order .....	D2-662
D2.2	Register restrictions for A64 instructions .....	D2-669
D2.3	ADC .....	D2-670
D2.4	ADCS .....	D2-671
D2.5	ADD (extended register) .....	D2-672
D2.6	ADD (immediate) .....	D2-674
D2.7	ADD (shifted register) .....	D2-675
D2.8	ADDG .....	D2-676

D2.9	ADDS (extended register) .....	D2-677
D2.10	ADDS (immediate) .....	D2-679
D2.11	ADDS (shifted register) .....	D2-680
D2.12	ADR .....	D2-681
D2.13	ADRP .....	D2-682
D2.14	AND (immediate) .....	D2-683
D2.15	AND (shifted register) .....	D2-684
D2.16	ANDS (immediate) .....	D2-685
D2.17	ANDS (shifted register) .....	D2-686
D2.18	ASR (register) .....	D2-687
D2.19	ASR (immediate) .....	D2-688
D2.20	ASRV .....	D2-689
D2.21	AT .....	D2-690
D2.22	AUTDA, AUTDZA .....	D2-692
D2.23	AUTDB, AUTDZB .....	D2-693
D2.24	AUTIA, AUTIZA, AUTIA1716, AUTIASP, AUTIAZ .....	D2-694
D2.25	AUTIB, AUTIZB, AUTIB1716, AUTIBSP, AUTIBZ .....	D2-695
D2.26	AXFlag .....	D2-696
D2.27	B.cond .....	D2-697
D2.28	B .....	D2-698
D2.29	BFC .....	D2-699
D2.30	BFI .....	D2-700
D2.31	BFM .....	D2-701
D2.32	BFXIL .....	D2-702
D2.33	BIC (shifted register) .....	D2-703
D2.34	BICS (shifted register) .....	D2-704
D2.35	BL .....	D2-705
D2.36	BLR .....	D2-706
D2.37	BLRAA, BLRAAZ, BLRAB, BLRABZ .....	D2-707
D2.38	BR .....	D2-708
D2.39	BRAA, BRAAZ, BRAB, BRABZ .....	D2-709
D2.40	BRK .....	D2-710
D2.41	BTI .....	D2-711
D2.42	CBNZ .....	D2-712
D2.43	CBZ .....	D2-713
D2.44	CCMN (immediate) .....	D2-714
D2.45	CCMN (register) .....	D2-715
D2.46	CCMP (immediate) .....	D2-716
D2.47	CCMP (register) .....	D2-717
D2.48	CINC .....	D2-718
D2.49	CINV .....	D2-719
D2.50	CLREX .....	D2-720
D2.51	CLS .....	D2-721
D2.52	CLZ .....	D2-722
D2.53	CMN (extended register) .....	D2-723
D2.54	CMN (immediate) .....	D2-725
D2.55	CMN (shifted register) .....	D2-726
D2.56	CMP (extended register) .....	D2-727
D2.57	CMP (immediate) .....	D2-729
D2.58	CMP (shifted register) .....	D2-730

D2.59	CMPP .....	D2-731
D2.60	CNEG .....	D2-732
D2.61	CRC32B, CRC32H, CRC32W, CRC32X .....	D2-733
D2.62	CRC32CB, CRC32CH, CRC32CW, CRC32CX .....	D2-734
D2.63	CSDB .....	D2-735
D2.64	CSEL .....	D2-737
D2.65	CSET .....	D2-738
D2.66	CSETM .....	D2-739
D2.67	CSINC .....	D2-740
D2.68	CSINV .....	D2-741
D2.69	CSNEG .....	D2-742
D2.70	DC .....	D2-743
D2.71	DCPS1 .....	D2-744
D2.72	DCPS2 .....	D2-745
D2.73	DCPS3 .....	D2-746
D2.74	DMB .....	D2-747
D2.75	DRPS .....	D2-749
D2.76	DSB .....	D2-750
D2.77	EON (shifted register) .....	D2-752
D2.78	EOR (immediate) .....	D2-753
D2.79	EOR (shifted register) .....	D2-754
D2.80	ERET .....	D2-755
D2.81	ERETAA, ERETAB .....	D2-756
D2.82	ESB .....	D2-757
D2.83	EXTR .....	D2-758
D2.84	GMI .....	D2-759
D2.85	HINT .....	D2-760
D2.86	HLT .....	D2-761
D2.87	HVC .....	D2-762
D2.88	IC .....	D2-763
D2.89	IRG .....	D2-764
D2.90	ISB .....	D2-765
D2.91	LDG .....	D2-766
D2.92	LDGV .....	D2-767
D2.93	LSL (register) .....	D2-768
D2.94	LSL (immediate) .....	D2-769
D2.95	LSLV .....	D2-770
D2.96	LSR (register) .....	D2-771
D2.97	LSR (immediate) .....	D2-772
D2.98	LSRV .....	D2-773
D2.99	MADD .....	D2-774
D2.100	MNEG .....	D2-775
D2.101	MOV (to or from SP) .....	D2-776
D2.102	MOV (inverted wide immediate) .....	D2-777
D2.103	MOV (wide immediate) .....	D2-778
D2.104	MOV (bitmask immediate) .....	D2-779
D2.105	MOV (register) .....	D2-780
D2.106	MOVK .....	D2-781
D2.107	MOVN .....	D2-782
D2.108	MOVZ .....	D2-783

D2.109	MRS .....	D2-784
D2.110	MSR ( <i>immediate</i> ) .....	D2-785
D2.111	MSR ( <i>register</i> ) .....	D2-786
D2.112	MSUB .....	D2-787
D2.113	MUL .....	D2-788
D2.114	MVN .....	D2-789
D2.115	NEG ( <i>shifted register</i> ) .....	D2-790
D2.116	NEGS .....	D2-791
D2.117	NGC .....	D2-792
D2.118	NGCS .....	D2-793
D2.119	NOP .....	D2-794
D2.120	ORN ( <i>shifted register</i> ) .....	D2-795
D2.121	ORR ( <i>immediate</i> ) .....	D2-796
D2.122	ORR ( <i>shifted register</i> ) .....	D2-797
D2.123	PACDA, PACDZA .....	D2-798
D2.124	PACDB, PACDZB .....	D2-799
D2.125	PACGA .....	D2-800
D2.126	PACIA, PACIZA, PACIA1716, PACIASP, PACIAZ .....	D2-801
D2.127	PACIB, PACIZB, PACIB1716, PACIBSP, PACIBZ .....	D2-802
D2.128	PSB .....	D2-803
D2.129	RBIT .....	D2-804
D2.130	RET .....	D2-805
D2.131	RETAAC, RETAB .....	D2-806
D2.132	REV16 .....	D2-807
D2.133	REV32 .....	D2-808
D2.134	REV64 .....	D2-809
D2.135	REV .....	D2-810
D2.136	ROR ( <i>immediate</i> ) .....	D2-811
D2.137	ROR ( <i>register</i> ) .....	D2-812
D2.138	RORV .....	D2-813
D2.139	SBC .....	D2-814
D2.140	SBCS .....	D2-815
D2.141	SBFIZ .....	D2-816
D2.142	SBFM .....	D2-817
D2.143	SBFX .....	D2-818
D2.144	SDIV .....	D2-819
D2.145	SEV .....	D2-820
D2.146	SEVL .....	D2-821
D2.147	SMADDL .....	D2-822
D2.148	SMC .....	D2-823
D2.149	SMNEGL .....	D2-824
D2.150	SMSUBL .....	D2-825
D2.151	SMULH .....	D2-826
D2.152	SMULL .....	D2-827
D2.153	ST2G .....	D2-828
D2.154	STG .....	D2-829
D2.155	STGP .....	D2-830
D2.156	STGV .....	D2-831
D2.157	STZ2G .....	D2-832
D2.158	STZG .....	D2-833

D2.159	SUB (extended register) .....	D2-834
D2.160	SUB (immediate) .....	D2-836
D2.161	SUB (shifted register) .....	D2-837
D2.162	SUBG .....	D2-838
D2.163	SUBP .....	D2-839
D2.164	SUBPS .....	D2-840
D2.165	SUBS (extended register) .....	D2-841
D2.166	SUBS (immediate) .....	D2-843
D2.167	SUBS (shifted register) .....	D2-844
D2.168	SVC .....	D2-845
D2.169	SXTB .....	D2-846
D2.170	SXTH .....	D2-847
D2.171	SXTW .....	D2-848
D2.172	SYS .....	D2-849
D2.173	SYSL .....	D2-850
D2.174	TBNZ .....	D2-851
D2.175	TBZ .....	D2-852
D2.176	TLBI .....	D2-853
D2.177	TST (immediate) .....	D2-855
D2.178	TST (shifted register) .....	D2-856
D2.179	UBFIZ .....	D2-857
D2.180	UBFM .....	D2-858
D2.181	UBFX .....	D2-859
D2.182	UDIV .....	D2-860
D2.183	UMADDL .....	D2-861
D2.184	UMNEGL .....	D2-862
D2.185	UMSUBL .....	D2-863
D2.186	UMULH .....	D2-864
D2.187	UMULL .....	D2-865
D2.188	UXTB .....	D2-866
D2.189	UXTH .....	D2-867
D2.190	XAFlag .....	D2-868
D2.191	WFE .....	D2-869
D2.192	WFI .....	D2-870
D2.193	XPACD, XPACI, XPAACLRI .....	D2-871
D2.194	YIELD .....	D2-872

## Chapter D3

### A64 Data Transfer Instructions

D3.1	A64 data transfer instructions in alphabetical order .....	D3-877
D3.2	CASA, CASAL, CAS, CASL, CASAL, CAS, CASL .....	D3-883
D3.3	CASAB, CASALB, CASB, CASLB .....	D3-884
D3.4	CASAH, CASALH, CASH, CASLH .....	D3-885
D3.5	CASPA, CASPAL, CASP, CASPL, CASPAL, CASP, CASPL .....	D3-886
D3.6	LDADDA, LDADDAL, LDADD, LDADDL, LDADDAL, LDADD, LDADDL .....	D3-888
D3.7	LDADDAB, LDADDALB, LDADDB, LDADDLB .....	D3-889
D3.8	LDADDAH, LDADDALH, LDADDH, LDADDLH .....	D3-890
D3.9	LDAPR .....	D3-891
D3.10	LDAPRB .....	D3-892
D3.11	LDAPRH .....	D3-893
D3.12	LDAR .....	D3-894

D3.13	<i>LDARB</i>	.....	D3-895
D3.14	<i>LDARH</i>	.....	D3-896
D3.15	<i>LDAXP</i>	.....	D3-897
D3.16	<i>LDAXR</i>	.....	D3-898
D3.17	<i>LDAXRB</i>	.....	D3-899
D3.18	<i>LDAXRH</i>	.....	D3-900
D3.19	<i>LDCLRA, LDCLRAL, LDCLR, LDCLRL, LDCLRAL, LDCLR, LDCLRL</i>	.....	D3-901
D3.20	<i>LDCLRAB, LDCLRALB, LDCLR, LDCLRLB</i>	.....	D3-902
D3.21	<i>LDCLRAH, LDCLRALH, LDCLR, LDCLRLH</i>	.....	D3-903
D3.22	<i>LDEORA, LDEORAL, LDEOR, LDEORL, LDEORAL, LDEOR, LDEORL</i>	.....	D3-904
D3.23	<i>LDEORAB, LDEORALB, LDEORB, LDEORLB</i>	.....	D3-905
D3.24	<i>LDEORAH, LDEORALH, LDEORH, LDEORLH</i>	.....	D3-906
D3.25	<i>LDLAR</i>	.....	D3-907
D3.26	<i>LDLARB</i>	.....	D3-908
D3.27	<i>LDLARH</i>	.....	D3-909
D3.28	<i>LDNP</i>	.....	D3-910
D3.29	<i>LDP</i>	.....	D3-911
D3.30	<i>LDPSW</i>	.....	D3-912
D3.31	<i>LDR (immediate)</i>	.....	D3-913
D3.32	<i>LDR (literal)</i>	.....	D3-914
D3.33	<i>LDR (register)</i>	.....	D3-915
D3.34	<i>LDRAA, LDRA, LDRA, LDRA</i>	.....	D3-916
D3.35	<i>LDRB (immediate)</i>	.....	D3-917
D3.36	<i>LDRB (register)</i>	.....	D3-918
D3.37	<i>LDRH (immediate)</i>	.....	D3-919
D3.38	<i>LDRH (register)</i>	.....	D3-920
D3.39	<i>LDRSB (immediate)</i>	.....	D3-921
D3.40	<i>LDRSB (register)</i>	.....	D3-922
D3.41	<i>LDRSH (immediate)</i>	.....	D3-923
D3.42	<i>LDRSH (register)</i>	.....	D3-924
D3.43	<i>LDRSW (immediate)</i>	.....	D3-925
D3.44	<i>LDRSW (literal)</i>	.....	D3-926
D3.45	<i>LDRSW (register)</i>	.....	D3-927
D3.46	<i>LDSETA, LDSETAL, LDSET, LDSETL, LDSETAL, LDSET, LDSETL</i>	.....	D3-928
D3.47	<i>LDSETAB, LDSETALB, LDSETB, LDSETLB</i>	.....	D3-929
D3.48	<i>LDSETAH, LDSETALH, LDSETH, LDSETLH</i>	.....	D3-930
D3.49	<i>LDSMAXA, LDSMAXAL, LDSMAX, LDSMAXL, LDSMAXAL, LDSMAX, LDSMAXL</i>	.....	D3-931
D3.50	<i>LDSMAXAB, LDSMAXALB, LDSMAXB, LDSMAXLB</i>	.....	D3-932
D3.51	<i>LDSMAXAH, LDSMAXALH, LDSMAXH, LDSMAXLH</i>	.....	D3-933
D3.52	<i>LDSMINA, LDSMINAL, LDSMIN, LDSMINL, LDSMINAL, LDSMIN, LDSMINL</i>	..	D3-934
D3.53	<i>LDSMINAB, LDSMINALB, LDSMINB, LDSMINLB</i>	.....	D3-935
D3.54	<i>LDSMINAH, LDSMINALH, LDSMINH, LDSMINLH</i>	.....	D3-936
D3.55	<i>LDTR</i>	.....	D3-937
D3.56	<i>LDTRB</i>	.....	D3-938
D3.57	<i>LDTRH</i>	.....	D3-939
D3.58	<i>LDTRSB</i>	.....	D3-940
D3.59	<i>LDTRSH</i>	.....	D3-941
D3.60	<i>LDTRSW</i>	.....	D3-942

D3.61	<i>LDUMAXA, LDUMAXAL, LDUMAX, LDUMAXL, LDUMAXAL, LDUMAX, LDUMAXL</i> ....	D3-943
D3.62	<i>LDUMAXAB, LDUMAXALB, LDUMAXB, LDUMAXLB</i> .....	D3-944
D3.63	<i>LDUMAXAH, LDUMAXALH, LDUMAXH, LDUMAXLH</i> .....	D3-945
D3.64	<i>LDUMINA, LDUMINAL, LDUMIN, LDUMINL, LDUMINAL, LDUMIN, LDUMINL</i> D3-946	
D3.65	<i>LDUMINAB, LDUMINALB, LDUMINB, LDUMINLB</i> .....	D3-947
D3.66	<i>LDUMINAH, LDUMINALH, LDUMINH, LDUMINLH</i> .....	D3-948
D3.67	<i>LDUR</i> .....	D3-949
D3.68	<i>LDURB</i> .....	D3-950
D3.69	<i>LDURH</i> .....	D3-951
D3.70	<i>LDURSB</i> .....	D3-952
D3.71	<i>LDURSH</i> .....	D3-953
D3.72	<i>LDURSW</i> .....	D3-954
D3.73	<i>LDXP</i> .....	D3-955
D3.74	<i>LDXR</i> .....	D3-956
D3.75	<i>LDXRB</i> .....	D3-957
D3.76	<i>LDXRH</i> .....	D3-958
D3.77	<i>PRFM (immediate)</i> .....	D3-959
D3.78	<i>PRFM (literal)</i> .....	D3-961
D3.79	<i>PRFM (register)</i> .....	D3-963
D3.80	<i>PRFUM (unscaled offset)</i> .....	D3-965
D3.81	<i>STADD, STADDL, STADDL</i> .....	D3-967
D3.82	<i>STADDB, STADDLB</i> .....	D3-968
D3.83	<i>STADDH, STADDLH</i> .....	D3-969
D3.84	<i>STCLR, STCLRL, STCLRL</i> .....	D3-970
D3.85	<i>STCLRB, STCLRLB</i> .....	D3-971
D3.86	<i>STCLRH, STCLRLH</i> .....	D3-972
D3.87	<i>STEOR, STEORL, STEORL</i> .....	D3-973
D3.88	<i>STEORB, STEORLB</i> .....	D3-974
D3.89	<i>STEORH, STEORLH</i> .....	D3-975
D3.90	<i>STLLR</i> .....	D3-976
D3.91	<i>STLLRB</i> .....	D3-977
D3.92	<i>STLLRH</i> .....	D3-978
D3.93	<i>STLR</i> .....	D3-979
D3.94	<i>STLRB</i> .....	D3-980
D3.95	<i>STLRH</i> .....	D3-981
D3.96	<i>STLXP</i> .....	D3-982
D3.97	<i>STLXR</i> .....	D3-984
D3.98	<i>STLXR</i> .....	D3-986
D3.99	<i>STLXRH</i> .....	D3-987
D3.100	<i>STNP</i> .....	D3-988
D3.101	<i>STP</i> .....	D3-989
D3.102	<i>STR (immediate)</i> .....	D3-990
D3.103	<i>STR (register)</i> .....	D3-991
D3.104	<i>STRB (immediate)</i> .....	D3-992
D3.105	<i>STRB (register)</i> .....	D3-993
D3.106	<i>STRH (immediate)</i> .....	D3-994
D3.107	<i>STRH (register)</i> .....	D3-995
D3.108	<i>STSET, STSETL, STSETL</i> .....	D3-996
D3.109	<i>STSETB, STSETLB</i> .....	D3-997

D3.110	STSETH, STSETLH .....	D3-998
D3.111	STSMAX, STSMAXL, STSMAXL .....	D3-999
D3.112	STSMAXB, STSMAXLB .....	D3-1000
D3.113	STSMAXH, STSMAXLH .....	D3-1001
D3.114	STSMIN, STSMINL, STSMINL .....	D3-1002
D3.115	STSMINB, STSMINLB .....	D3-1003
D3.116	STSMINH, STSMINLH .....	D3-1004
D3.117	STTR .....	D3-1005
D3.118	STTRB .....	D3-1006
D3.119	STTRH .....	D3-1007
D3.120	STUMAX, STUMAXL, STUMAXL .....	D3-1008
D3.121	STUMAXB, STUMAXLB .....	D3-1009
D3.122	STUMAXH, STUMAXLH .....	D3-1010
D3.123	STUMIN, STUMINL, STUMINL .....	D3-1011
D3.124	STUMINB, STUMINLB .....	D3-1012
D3.125	STUMINH, STUMINLH .....	D3-1013
D3.126	STUR .....	D3-1014
D3.127	STURB .....	D3-1015
D3.128	STURH .....	D3-1016
D3.129	STXP .....	D3-1017
D3.130	STXR .....	D3-1019
D3.131	STXRB .....	D3-1021
D3.132	STXRH .....	D3-1022
D3.133	SWPA, SWPAL, SWP, SWPL, SWPAL, SWP, SWPL .....	D3-1023
D3.134	SWPAB, SWPALB, SWPB, SWPLB .....	D3-1024
D3.135	SWPAH, SWPALH, SWPH, SWPLH .....	D3-1025

## Chapter D4

### A64 Floating-point Instructions

D4.1	A64 floating-point instructions in alphabetical order .....	D4-1029
D4.2	Register restrictions for A64 instructions .....	D4-1032
D4.3	FABS (scalar) .....	D4-1033
D4.4	FADD (scalar) .....	D4-1034
D4.5	FCCMP .....	D4-1035
D4.6	FCCMPE .....	D4-1036
D4.7	FCMP .....	D4-1038
D4.8	FCMPE .....	D4-1040
D4.9	FCSEL .....	D4-1042
D4.10	FCVT .....	D4-1043
D4.11	FCVTA (scalar) .....	D4-1044
D4.12	FCVTAU (scalar) .....	D4-1045
D4.13	FCVTMS (scalar) .....	D4-1046
D4.14	FCVTPU (scalar) .....	D4-1047
D4.15	FCVTNS (scalar) .....	D4-1048
D4.16	FCVTNU (scalar) .....	D4-1049
D4.17	FCVTPS (scalar) .....	D4-1050
D4.18	FCVTPU (scalar) .....	D4-1051
D4.19	FCVTZS (scalar, fixed-point) .....	D4-1052
D4.20	FCVTZS (scalar, integer) .....	D4-1054
D4.21	FCVTZU (scalar, fixed-point) .....	D4-1055
D4.22	FCVTZU (scalar, integer) .....	D4-1057

D4.23	<i>FDIV (scalar)</i>	.....	D4-1058
D4.24	<i>FJCVTZS</i>	.....	D4-1059
D4.25	<i>FMADD</i>	.....	D4-1060
D4.26	<i>FMAX (scalar)</i>	.....	D4-1061
D4.27	<i>FMAXNM (scalar)</i>	.....	D4-1062
D4.28	<i>FMIN (scalar)</i>	.....	D4-1063
D4.29	<i>FMINNM (scalar)</i>	.....	D4-1064
D4.30	<i>FMOV (register)</i>	.....	D4-1065
D4.31	<i>FMOV (general)</i>	.....	D4-1066
D4.32	<i>FMOV (scalar, immediate)</i>	.....	D4-1067
D4.33	<i>FMSUB</i>	.....	D4-1068
D4.34	<i>FMUL (scalar)</i>	.....	D4-1069
D4.35	<i>FNEG (scalar)</i>	.....	D4-1070
D4.36	<i>FNMADD</i>	.....	D4-1071
D4.37	<i>FNMSUB</i>	.....	D4-1072
D4.38	<i>FNMUL (scalar)</i>	.....	D4-1073
D4.39	<i>FRINTA (scalar)</i>	.....	D4-1074
D4.40	<i>FRINTI (scalar)</i>	.....	D4-1075
D4.41	<i>FRINTM (scalar)</i>	.....	D4-1076
D4.42	<i>FRINTN (scalar)</i>	.....	D4-1077
D4.43	<i>FRINTP (scalar)</i>	.....	D4-1078
D4.44	<i>FRINTX (scalar)</i>	.....	D4-1079
D4.45	<i>FRINTZ (scalar)</i>	.....	D4-1080
D4.46	<i>FSQRT (scalar)</i>	.....	D4-1081
D4.47	<i>FSUB (scalar)</i>	.....	D4-1082
D4.48	<i>LDNP (SIMD and FP)</i>	.....	D4-1083
D4.49	<i>LDP (SIMD and FP)</i>	.....	D4-1085
D4.50	<i>LDR (immediate, SIMD and FP)</i>	.....	D4-1087
D4.51	<i>LDR (literal, SIMD and FP)</i>	.....	D4-1089
D4.52	<i>LDR (register, SIMD and FP)</i>	.....	D4-1090
D4.53	<i>LDUR (SIMD and FP)</i>	.....	D4-1092
D4.54	<i>SCVTF (scalar, fixed-point)</i>	.....	D4-1093
D4.55	<i>SCVTF (scalar, integer)</i>	.....	D4-1095
D4.56	<i>STNP (SIMD and FP)</i>	.....	D4-1096
D4.57	<i>STP (SIMD and FP)</i>	.....	D4-1097
D4.58	<i>STR (immediate, SIMD and FP)</i>	.....	D4-1098
D4.59	<i>STR (register, SIMD and FP)</i>	.....	D4-1100
D4.60	<i>STUR (SIMD and FP)</i>	.....	D4-1102
D4.61	<i>UCVTF (scalar, fixed-point)</i>	.....	D4-1103
D4.62	<i>UCVTF (scalar, integer)</i>	.....	D4-1105

## Chapter D5

### A64 SIMD Scalar Instructions

D5.1	<i>A64 SIMD scalar instructions in alphabetical order</i>	.....	D5-1110
D5.2	<i>ABS (scalar)</i>	.....	D5-1115
D5.3	<i>ADD (scalar)</i>	.....	D5-1116
D5.4	<i>ADDP (scalar)</i>	.....	D5-1117
D5.5	<i>CMEQ (scalar, register)</i>	.....	D5-1118
D5.6	<i>CMEQ (scalar, zero)</i>	.....	D5-1119
D5.7	<i>CMGE (scalar, register)</i>	.....	D5-1120
D5.8	<i>CMGE (scalar, zero)</i>	.....	D5-1121

D5.9	<i>CMGT (scalar, register)</i>	D5-1122
D5.10	<i>CMGT (scalar, zero)</i>	D5-1123
D5.11	<i>CMHI (scalar, register)</i>	D5-1124
D5.12	<i>CMHS (scalar, register)</i>	D5-1125
D5.13	<i>CMLE (scalar, zero)</i>	D5-1126
D5.14	<i>CMLT (scalar, zero)</i>	D5-1127
D5.15	<i>CMTST (scalar)</i>	D5-1128
D5.16	<i>DUP (scalar, element)</i>	D5-1129
D5.17	<i>FABD (scalar)</i>	D5-1130
D5.18	<i>FACGE (scalar)</i>	D5-1131
D5.19	<i>FACGT (scalar)</i>	D5-1132
D5.20	<i>FADDP (scalar)</i>	D5-1133
D5.21	<i>FCMEQ (scalar, register)</i>	D5-1134
D5.22	<i>FCMEQ (scalar, zero)</i>	D5-1135
D5.23	<i>FCMGE (scalar, register)</i>	D5-1136
D5.24	<i>FCMGE (scalar, zero)</i>	D5-1137
D5.25	<i>FCMGT (scalar, register)</i>	D5-1138
D5.26	<i>FCMGT (scalar, zero)</i>	D5-1139
D5.27	<i>FCMLA (scalar, by element)</i>	D5-1140
D5.28	<i>FCMLE (scalar, zero)</i>	D5-1142
D5.29	<i>FCMLT (scalar, zero)</i>	D5-1143
D5.30	<i>FCVTAS (scalar)</i>	D5-1144
D5.31	<i>FCVTAU (scalar)</i>	D5-1145
D5.32	<i>FCVTMS (scalar)</i>	D5-1146
D5.33	<i>FCVTMU (scalar)</i>	D5-1147
D5.34	<i>FCVTNS (scalar)</i>	D5-1148
D5.35	<i>FCVTNU (scalar)</i>	D5-1149
D5.36	<i>FCVTPS (scalar)</i>	D5-1150
D5.37	<i>FCVTPU (scalar)</i>	D5-1151
D5.38	<i>FCVTXN (scalar)</i>	D5-1152
D5.39	<i>FCVTZS (scalar, fixed-point)</i>	D5-1153
D5.40	<i>FCVTZS (scalar, integer)</i>	D5-1154
D5.41	<i>FCVTZU (scalar, fixed-point)</i>	D5-1155
D5.42	<i>FCVTZU (scalar, integer)</i>	D5-1156
D5.43	<i>FMAXNMP (scalar)</i>	D5-1157
D5.44	<i>FMAXP (scalar)</i>	D5-1158
D5.45	<i>FMINNMP (scalar)</i>	D5-1159
D5.46	<i>FMINP (scalar)</i>	D5-1160
D5.47	<i>FMLA (scalar, by element)</i>	D5-1161
D5.48	<i>FMLAL, (scalar, by element)</i>	D5-1163
D5.49	<i>FMLS (scalar, by element)</i>	D5-1164
D5.50	<i>FMLSL, (scalar, by element)</i>	D5-1166
D5.51	<i>FMUL (scalar, by element)</i>	D5-1167
D5.52	<i>FMULX (scalar, by element)</i>	D5-1169
D5.53	<i>FMULX (scalar)</i>	D5-1171
D5.54	<i>FRECPE (scalar)</i>	D5-1172
D5.55	<i>FRECPSEN (scalar)</i>	D5-1173
D5.56	<i>FRSQRTS (scalar)</i>	D5-1174
D5.57	<i>FRSQRTS (scalar)</i>	D5-1175
D5.58	<i>MOV (scalar)</i>	D5-1176

D5.59	NEG (scalar) .....	D5-1177
D5.60	SCVT <sub>F</sub> (scalar, fixed-point) .....	D5-1178
D5.61	SCVT <sub>F</sub> (scalar, integer) .....	D5-1179
D5.62	SHL (scalar) .....	D5-1180
D5.63	SLI (scalar) .....	D5-1181
D5.64	SQABS (scalar) .....	D5-1182
D5.65	SQADD (scalar) .....	D5-1183
D5.66	SQDMLAL (scalar, by element) .....	D5-1184
D5.67	SQDMLAL (scalar) .....	D5-1185
D5.68	SQDMLSL (scalar, by element) .....	D5-1186
D5.69	SQDMLSL (scalar) .....	D5-1187
D5.70	SQDMULH (scalar, by element) .....	D5-1188
D5.71	SQDMULH (scalar) .....	D5-1189
D5.72	SQDMULL (scalar, by element) .....	D5-1190
D5.73	SQDMULL (scalar) .....	D5-1191
D5.74	SQNEG (scalar) .....	D5-1192
D5.75	SQRDMLAH (scalar, by element) .....	D5-1193
D5.76	SQRDMLAH (scalar) .....	D5-1194
D5.77	SQRDMLSH (scalar, by element) .....	D5-1195
D5.78	SQRDMLSH (scalar) .....	D5-1196
D5.79	SQRDMULH (scalar, by element) .....	D5-1197
D5.80	SQRDMULH (scalar) .....	D5-1198
D5.81	SQRSHL (scalar) .....	D5-1199
D5.82	SQRSHRN (scalar) .....	D5-1200
D5.83	SQRSHRUN (scalar) .....	D5-1201
D5.84	SQSHL (scalar, immediate) .....	D5-1202
D5.85	SQSHL (scalar, register) .....	D5-1203
D5.86	SQSHLU (scalar) .....	D5-1204
D5.87	SQSHRN (scalar) .....	D5-1205
D5.88	SQSHRUN (scalar) .....	D5-1206
D5.89	SQSUB (scalar) .....	D5-1207
D5.90	SQXTN (scalar) .....	D5-1208
D5.91	SQXTUN (scalar) .....	D5-1209
D5.92	SRI (scalar) .....	D5-1210
D5.93	SRSHL (scalar) .....	D5-1211
D5.94	SRSHR (scalar) .....	D5-1212
D5.95	SRSRA (scalar) .....	D5-1213
D5.96	SSH <sub>L</sub> (scalar) .....	D5-1214
D5.97	SSH <sub>R</sub> (scalar) .....	D5-1215
D5.98	SSRA (scalar) .....	D5-1216
D5.99	SUB (scalar) .....	D5-1217
D5.100	SUQADD (scalar) .....	D5-1218
D5.101	UCVT <sub>F</sub> (scalar, fixed-point) .....	D5-1219
D5.102	UCVT <sub>F</sub> (scalar, integer) .....	D5-1220
D5.103	UQADD (scalar) .....	D5-1221
D5.104	UQRSHL (scalar) .....	D5-1222
D5.105	UQRSHRN (scalar) .....	D5-1223
D5.106	UQSHL (scalar, immediate) .....	D5-1224
D5.107	UQSHL (scalar, register) .....	D5-1225
D5.108	UQSHRN (scalar) .....	D5-1226

D5.109	<i>UQSUB (scalar)</i>	D5-1227
D5.110	<i>UQXTN (scalar)</i>	D5-1228
D5.111	<i>URSHL (scalar)</i>	D5-1229
D5.112	<i>URSHR (scalar)</i>	D5-1230
D5.113	<i>URSRA (scalar)</i>	D5-1231
D5.114	<i>USHL (scalar)</i>	D5-1232
D5.115	<i>USHR (scalar)</i>	D5-1233
D5.116	<i>USQADD (scalar)</i>	D5-1234
D5.117	<i>USRA (scalar)</i>	D5-1235

## Chapter D6

### A64 SIMD Vector Instructions

D6.1	<i>A64 SIMD Vector instructions in alphabetical order</i>	D6-1243
D6.2	<i>ABS (vector)</i>	D6-1254
D6.3	<i>ADD (vector)</i>	D6-1255
D6.4	<i>ADDHN, ADDHN2 (vector)</i>	D6-1256
D6.5	<i>ADDP (vector)</i>	D6-1257
D6.6	<i>ADDV (vector)</i>	D6-1258
D6.7	<i>AND (vector)</i>	D6-1259
D6.8	<i>BIC (vector, immediate)</i>	D6-1260
D6.9	<i>BIC (vector, register)</i>	D6-1261
D6.10	<i>BIF (vector)</i>	D6-1262
D6.11	<i>BIT (vector)</i>	D6-1263
D6.12	<i>BSL (vector)</i>	D6-1264
D6.13	<i>CLS (vector)</i>	D6-1265
D6.14	<i>CLZ (vector)</i>	D6-1266
D6.15	<i>CMEQ (vector, register)</i>	D6-1267
D6.16	<i>CMEQ (vector, zero)</i>	D6-1268
D6.17	<i>CMGE (vector, register)</i>	D6-1269
D6.18	<i>CMGE (vector, zero)</i>	D6-1270
D6.19	<i>CMGT (vector, register)</i>	D6-1271
D6.20	<i>CMGT (vector, zero)</i>	D6-1272
D6.21	<i>CMHI (vector, register)</i>	D6-1273
D6.22	<i>CMHS (vector, register)</i>	D6-1274
D6.23	<i>CMLE (vector, zero)</i>	D6-1275
D6.24	<i>CMLT (vector, zero)</i>	D6-1276
D6.25	<i>CMTST (vector)</i>	D6-1277
D6.26	<i>CNT (vector)</i>	D6-1278
D6.27	<i>DUP (vector, element)</i>	D6-1279
D6.28	<i>DUP (vector, general)</i>	D6-1280
D6.29	<i>EOR (vector)</i>	D6-1281
D6.30	<i>EXT (vector)</i>	D6-1282
D6.31	<i>FABD (vector)</i>	D6-1283
D6.32	<i>FABS (vector)</i>	D6-1284
D6.33	<i>FACGE (vector)</i>	D6-1285
D6.34	<i>FACGT (vector)</i>	D6-1286
D6.35	<i>FADD (vector)</i>	D6-1287
D6.36	<i>FADDP (vector)</i>	D6-1288
D6.37	<i>FCADD (vector)</i>	D6-1289
D6.38	<i>FCMEQ (vector, register)</i>	D6-1290
D6.39	<i>FCMEQ (vector, zero)</i>	D6-1291

D6.40	<i>FCMGE (vector, register)</i>	D6-1292
D6.41	<i>FCMGE (vector, zero)</i>	D6-1293
D6.42	<i>FCMGT (vector, register)</i>	D6-1294
D6.43	<i>FCMGT (vector, zero)</i>	D6-1295
D6.44	<i>FCMLA (vector)</i>	D6-1296
D6.45	<i>FCMLE (vector, zero)</i>	D6-1297
D6.46	<i>FCMLT (vector, zero)</i>	D6-1298
D6.47	<i>FCVTAS (vector)</i>	D6-1299
D6.48	<i>FCVTAU (vector)</i>	D6-1300
D6.49	<i>FCVTL, FCVTL2 (vector)</i>	D6-1301
D6.50	<i>FCVTPS (vector)</i>	D6-1302
D6.51	<i>FCVTPU (vector)</i>	D6-1303
D6.52	<i>FCVTN, FCVTN2 (vector)</i>	D6-1304
D6.53	<i>FCVTNS (vector)</i>	D6-1305
D6.54	<i>FCVTNU (vector)</i>	D6-1306
D6.55	<i>FCVTPS (vector)</i>	D6-1307
D6.56	<i>FCVTPU (vector)</i>	D6-1308
D6.57	<i>FCVTXN, FCVTXN2 (vector)</i>	D6-1309
D6.58	<i>FCVTZS (vector, fixed-point)</i>	D6-1310
D6.59	<i>FCVTZS (vector, integer)</i>	D6-1311
D6.60	<i>FCVTZU (vector, fixed-point)</i>	D6-1312
D6.61	<i>FCVTZU (vector, integer)</i>	D6-1313
D6.62	<i>FDIV (vector)</i>	D6-1314
D6.63	<i>FMAX (vector)</i>	D6-1315
D6.64	<i>FMAXNM (vector)</i>	D6-1316
D6.65	<i>FMAXNMP (vector)</i>	D6-1317
D6.66	<i>FMAXNMV (vector)</i>	D6-1318
D6.67	<i>FMAXP (vector)</i>	D6-1319
D6.68	<i>FMAXV (vector)</i>	D6-1320
D6.69	<i>FMIN (vector)</i>	D6-1321
D6.70	<i>FMINNM (vector)</i>	D6-1322
D6.71	<i>FMINNMP (vector)</i>	D6-1323
D6.72	<i>FMINNMV (vector)</i>	D6-1324
D6.73	<i>FMINP (vector)</i>	D6-1325
D6.74	<i>FMINV (vector)</i>	D6-1326
D6.75	<i>FMLA (vector, by element)</i>	D6-1327
D6.76	<i>FMLA (vector)</i>	D6-1329
D6.77	<i>FMLAL, (vector)</i>	D6-1330
D6.78	<i>FMLS (vector, by element)</i>	D6-1331
D6.79	<i>FMLS (vector)</i>	D6-1333
D6.80	<i>FMLSL, (vector)</i>	D6-1334
D6.81	<i>FMOV (vector, immediate)</i>	D6-1335
D6.82	<i>FMUL (vector, by element)</i>	D6-1337
D6.83	<i>FMUL (vector)</i>	D6-1339
D6.84	<i>FMULX (vector, by element)</i>	D6-1340
D6.85	<i>FMULX (vector)</i>	D6-1342
D6.86	<i>FNEG (vector)</i>	D6-1343
D6.87	<i>FRECPE (vector)</i>	D6-1344
D6.88	<i>FRECPS (vector)</i>	D6-1345
D6.89	<i>FRECPX (vector)</i>	D6-1346

<i>D6.90 FRINTA (vector)</i> .....	<i>D6-1347</i>
<i>D6.91 FRINTI (vector)</i> .....	<i>D6-1348</i>
<i>D6.92 FRINTM (vector)</i> .....	<i>D6-1349</i>
<i>D6.93 FRINTN (vector)</i> .....	<i>D6-1350</i>
<i>D6.94 FRINTP (vector)</i> .....	<i>D6-1351</i>
<i>D6.95 FRINTX (vector)</i> .....	<i>D6-1352</i>
<i>D6.96 FRINTZ (vector)</i> .....	<i>D6-1353</i>
<i>D6.97 FRSQRTE (vector)</i> .....	<i>D6-1354</i>
<i>D6.98 FRSQRTS (vector)</i> .....	<i>D6-1355</i>
<i>D6.99 FSQRT (vector)</i> .....	<i>D6-1356</i>
<i>D6.100 FSUB (vector)</i> .....	<i>D6-1357</i>
<i>D6.101 INS (vector, element)</i> .....	<i>D6-1358</i>
<i>D6.102 INS (vector, general)</i> .....	<i>D6-1359</i>
<i>D6.103 LD1 (vector, multiple structures)</i> .....	<i>D6-1360</i>
<i>D6.104 LD1 (vector, single structure)</i> .....	<i>D6-1363</i>
<i>D6.105 LD1R (vector)</i> .....	<i>D6-1364</i>
<i>D6.106 LD2 (vector, multiple structures)</i> .....	<i>D6-1365</i>
<i>D6.107 LD2 (vector, single structure)</i> .....	<i>D6-1366</i>
<i>D6.108 LD2R (vector)</i> .....	<i>D6-1367</i>
<i>D6.109 LD3 (vector, multiple structures)</i> .....	<i>D6-1368</i>
<i>D6.110 LD3 (vector, single structure)</i> .....	<i>D6-1369</i>
<i>D6.111 LD3R (vector)</i> .....	<i>D6-1371</i>
<i>D6.112 LD4 (vector, multiple structures)</i> .....	<i>D6-1372</i>
<i>D6.113 LD4 (vector, single structure)</i> .....	<i>D6-1373</i>
<i>D6.114 LD4R (vector)</i> .....	<i>D6-1375</i>
<i>D6.115 MLA (vector, by element)</i> .....	<i>D6-1376</i>
<i>D6.116 MLA (vector)</i> .....	<i>D6-1377</i>
<i>D6.117 MLS (vector, by element)</i> .....	<i>D6-1378</i>
<i>D6.118 MLS (vector)</i> .....	<i>D6-1379</i>
<i>D6.119 MOV (vector, element)</i> .....	<i>D6-1380</i>
<i>D6.120 MOV (vector, from general)</i> .....	<i>D6-1381</i>
<i>D6.121 MOV (vector)</i> .....	<i>D6-1382</i>
<i>D6.122 MOV (vector, to general)</i> .....	<i>D6-1383</i>
<i>D6.123 MOVI (vector)</i> .....	<i>D6-1384</i>
<i>D6.124 MUL (vector, by element)</i> .....	<i>D6-1386</i>
<i>D6.125 MUL (vector)</i> .....	<i>D6-1387</i>
<i>D6.126 MVN (vector)</i> .....	<i>D6-1388</i>
<i>D6.127 MVNI (vector)</i> .....	<i>D6-1389</i>
<i>D6.128 NEG (vector)</i> .....	<i>D6-1390</i>
<i>D6.129 NOT (vector)</i> .....	<i>D6-1391</i>
<i>D6.130 ORN (vector)</i> .....	<i>D6-1392</i>
<i>D6.131 ORR (vector, immediate)</i> .....	<i>D6-1393</i>
<i>D6.132 ORR (vector, register)</i> .....	<i>D6-1394</i>
<i>D6.133 PMUL (vector)</i> .....	<i>D6-1395</i>
<i>D6.134 PMULL, PMULL2 (vector)</i> .....	<i>D6-1396</i>
<i>D6.135 RADDHN, RADDHN2 (vector)</i> .....	<i>D6-1397</i>
<i>D6.136 RBIT (vector)</i> .....	<i>D6-1398</i>
<i>D6.137 REV16 (vector)</i> .....	<i>D6-1399</i>
<i>D6.138 REV32 (vector)</i> .....	<i>D6-1400</i>
<i>D6.139 REV64 (vector)</i> .....	<i>D6-1401</i>

D6.140	RSHRN, RSHRN2 (vector) .....	D6-1402
D6.141	RSUBHN, RSUBHN2 (vector) .....	D6-1403
D6.142	SABA (vector) .....	D6-1404
D6.143	SABAL, SABAL2 (vector) .....	D6-1405
D6.144	SABD (vector) .....	D6-1406
D6.145	SABDL, SABDL2 (vector) .....	D6-1407
D6.146	SADALP (vector) .....	D6-1408
D6.147	SADDL, SADDL2 (vector) .....	D6-1409
D6.148	SADDLP (vector) .....	D6-1410
D6.149	SADDLV (vector) .....	D6-1411
D6.150	SADDW, SADDW2 (vector) .....	D6-1412
D6.151	SCVTF (vector, fixed-point) .....	D6-1413
D6.152	SCVTF (vector, integer) .....	D6-1414
D6.153	SDOT (vector, by element) .....	D6-1415
D6.154	SDOT (vector) .....	D6-1416
D6.155	SHADD (vector) .....	D6-1417
D6.156	SHL (vector) .....	D6-1418
D6.157	SHLL, SHLL2 (vector) .....	D6-1419
D6.158	SHRN, SHRN2 (vector) .....	D6-1420
D6.159	SHSUB (vector) .....	D6-1421
D6.160	SLI (vector) .....	D6-1422
D6.161	SMAX (vector) .....	D6-1423
D6.162	SMAXP (vector) .....	D6-1424
D6.163	SMAXV (vector) .....	D6-1425
D6.164	SMIN (vector) .....	D6-1426
D6.165	SMINP (vector) .....	D6-1427
D6.166	SMINV (vector) .....	D6-1428
D6.167	SMLAL, SMLAL2 (vector, by element) .....	D6-1429
D6.168	SMLAL, SMLAL2 (vector) .....	D6-1430
D6.169	SMLS L, SMLS L2 (vector, by element) .....	D6-1431
D6.170	SMLS L, SMLS L2 (vector) .....	D6-1432
D6.171	SMOV (vector) .....	D6-1433
D6.172	SMULL, SMULL2 (vector, by element) .....	D6-1434
D6.173	SMULL, SMULL2 (vector) .....	D6-1435
D6.174	SQABS (vector) .....	D6-1436
D6.175	SQADD (vector) .....	D6-1437
D6.176	SQDMLAL, SQDMLAL2 (vector, by element) .....	D6-1438
D6.177	SQDMLAL, SQDMLAL2 (vector) .....	D6-1440
D6.178	SQDMLSL, SQDMLSL2 (vector, by element) .....	D6-1441
D6.179	SQDMLSL, SQDMLSL2 (vector) .....	D6-1443
D6.180	SQDMULH (vector, by element) .....	D6-1444
D6.181	SQDMULH (vector) .....	D6-1445
D6.182	SQDMULL, SQDMULL2 (vector, by element) .....	D6-1446
D6.183	SQDMULL, SQDMULL2 (vector) .....	D6-1448
D6.184	SQNEG (vector) .....	D6-1449
D6.185	SQRDMLAH (vector, by element) .....	D6-1450
D6.186	SQRDMLAH (vector) .....	D6-1451
D6.187	SQRDMLSH (vector, by element) .....	D6-1452
D6.188	SQRDMLSH (vector) .....	D6-1453
D6.189	SQRDMULH (vector, by element) .....	D6-1454

D6.190	SQRDMULH (vector)	.....	D6-1455
D6.191	SQRSHL (vector)	.....	D6-1456
D6.192	SQRSHRN, SQRSHRN2 (vector)	.....	D6-1457
D6.193	SQRSHRUN, SQRSHRUN2 (vector)	.....	D6-1458
D6.194	SQSHL (vector, immediate)	.....	D6-1459
D6.195	SQSHL (vector, register)	.....	D6-1460
D6.196	SQSHLU (vector)	.....	D6-1461
D6.197	SQSHRN, SQSHRN2 (vector)	.....	D6-1462
D6.198	SQSHRUN, SQSHRUN2 (vector)	.....	D6-1463
D6.199	SQSUB (vector)	.....	D6-1464
D6.200	SQXTN, SQXTN2 (vector)	.....	D6-1465
D6.201	SQXTUN, SQXTUN2 (vector)	.....	D6-1466
D6.202	SRHADD (vector)	.....	D6-1467
D6.203	SRI (vector)	.....	D6-1468
D6.204	SRSHL (vector)	.....	D6-1469
D6.205	SRSHR (vector)	.....	D6-1470
D6.206	SRSRA (vector)	.....	D6-1471
D6.207	SSHLL (vector)	.....	D6-1472
D6.208	SSHLL, SSHLL2 (vector)	.....	D6-1473
D6.209	SSHR (vector)	.....	D6-1474
D6.210	SSRA (vector)	.....	D6-1475
D6.211	SSUBL, SSUBL2 (vector)	.....	D6-1476
D6.212	SSUBW, SSUBW2 (vector)	.....	D6-1477
D6.213	ST1 (vector, multiple structures)	.....	D6-1478
D6.214	ST1 (vector, single structure)	.....	D6-1481
D6.215	ST2 (vector, multiple structures)	.....	D6-1482
D6.216	ST2 (vector, single structure)	.....	D6-1483
D6.217	ST3 (vector, multiple structures)	.....	D6-1484
D6.218	ST3 (vector, single structure)	.....	D6-1485
D6.219	ST4 (vector, multiple structures)	.....	D6-1487
D6.220	ST4 (vector, single structure)	.....	D6-1488
D6.221	SUB (vector)	.....	D6-1490
D6.222	SUBHN, SUBHN2 (vector)	.....	D6-1491
D6.223	SUQADD (vector)	.....	D6-1492
D6.224	SXTL, SXTL2 (vector)	.....	D6-1493
D6.225	TBL (vector)	.....	D6-1494
D6.226	TBX (vector)	.....	D6-1495
D6.227	TRN1 (vector)	.....	D6-1496
D6.228	TRN2 (vector)	.....	D6-1497
D6.229	UABA (vector)	.....	D6-1498
D6.230	UABAL, UABAL2 (vector)	.....	D6-1499
D6.231	UABD (vector)	.....	D6-1500
D6.232	UABDL, UABDL2 (vector)	.....	D6-1501
D6.233	UADALP (vector)	.....	D6-1502
D6.234	UADDL, UADDL2 (vector)	.....	D6-1503
D6.235	UADDLP (vector)	.....	D6-1504
D6.236	UADDLV (vector)	.....	D6-1505
D6.237	UADDW, UADDW2 (vector)	.....	D6-1506
D6.238	UCVTF (vector, fixed-point)	.....	D6-1507
D6.239	UCVTF (vector, integer)	.....	D6-1508

D6.240	UDOT (vector, by element) .....	D6-1509
D6.241	UDOT (vector) .....	D6-1510
D6.242	UHADD (vector) .....	D6-1511
D6.243	UHSUB (vector) .....	D6-1512
D6.244	UMAX (vector) .....	D6-1513
D6.245	UMAXP (vector) .....	D6-1514
D6.246	UMAXV (vector) .....	D6-1515
D6.247	UMIN (vector) .....	D6-1516
D6.248	UMINP (vector) .....	D6-1517
D6.249	UMINV (vector) .....	D6-1518
D6.250	UMLAL, UMLAL2 (vector, by element) .....	D6-1519
D6.251	UMLAL, UMLAL2 (vector) .....	D6-1520
D6.252	UMLSL, UMLSL2 (vector, by element) .....	D6-1521
D6.253	UMLSL, UMLSL2 (vector) .....	D6-1522
D6.254	UMOV (vector) .....	D6-1523
D6.255	UMULL, UMULL2 (vector, by element) .....	D6-1524
D6.256	UMULL, UMULL2 (vector) .....	D6-1525
D6.257	UQADD (vector) .....	D6-1526
D6.258	UQRSHL (vector) .....	D6-1527
D6.259	UQRSHRN, UQRSHRN2 (vector) .....	D6-1528
D6.260	UQSHL (vector, immediate) .....	D6-1529
D6.261	UQSHL (vector, register) .....	D6-1530
D6.262	UQSHRN, UQSHRN2 (vector) .....	D6-1531
D6.263	UQSUB (vector) .....	D6-1533
D6.264	UQXTN, UQXTN2 (vector) .....	D6-1534
D6.265	URECPE (vector) .....	D6-1535
D6.266	URHADD (vector) .....	D6-1536
D6.267	URSHL (vector) .....	D6-1537
D6.268	URSHR (vector) .....	D6-1538
D6.269	URSQRTE (vector) .....	D6-1539
D6.270	URSRA (vector) .....	D6-1540
D6.271	USHL (vector) .....	D6-1541
D6.272	USHLL, USHLL2 (vector) .....	D6-1542
D6.273	USHR (vector) .....	D6-1543
D6.274	USQADD (vector) .....	D6-1544
D6.275	USRA (vector) .....	D6-1545
D6.276	USUBL, USUBL2 (vector) .....	D6-1546
D6.277	USUBW, USUBW2 (vector) .....	D6-1547
D6.278	UXTL, UXTL2 (vector) .....	D6-1548
D6.279	UZP1 (vector) .....	D6-1549
D6.280	UZP2 (vector) .....	D6-1550
D6.281	XTN, XTN2 (vector) .....	D6-1551
D6.282	ZIP1 (vector) .....	D6-1552
D6.283	ZIP2 (vector) .....	D6-1553

## Chapter D7

## A64 Cryptographic Algorithms

D7.1	A64 Cryptographic instructions .....	D7-1556
------	--------------------------------------	---------



# List of Figures

## Arm® Instruction Set Reference Guide

<i>Figure A2-1</i>	<i>Organization of general-purpose registers and Program Status Registers .....</i>	A2-58
<i>Figure B1-1</i>	<i>Extension register bank for Advanced SIMD in AArch32 state .....</i>	B1-89
<i>Figure B1-2</i>	<i>Extension register bank for Advanced SIMD in AArch64 state .....</i>	B1-91
<i>Figure B2-1</i>	<i>Extension register bank for floating-point in AArch32 state .....</i>	B2-115
<i>Figure B2-2</i>	<i>Extension register bank for floating-point in AArch64 state .....</i>	B2-117
<i>Figure C2-1</i>	<i>ASR #3 .....</i>	C2-165
<i>Figure C2-2</i>	<i>LSR #3 .....</i>	C2-166
<i>Figure C2-3</i>	<i>LSL #3 .....</i>	C2-166
<i>Figure C2-4</i>	<i>ROR #3 .....</i>	C2-166
<i>Figure C2-5</i>	<i>RRX .....</i>	C2-167
<i>Figure C3-1</i>	<i>De-interleaving an array of 3-element structures .....</i>	C3-449
<i>Figure C3-2</i>	<i>Operation of doubleword VEXT for imm = 3 .....</i>	C3-489
<i>Figure C3-3</i>	<i>Example of operation of VPADAL (in this case for data type S16) .....</i>	C3-537
<i>Figure C3-4</i>	<i>Example of operation of VPADD (in this case, for data type I16) .....</i>	C3-538
<i>Figure C3-5</i>	<i>Example of operation of doubleword VPADDL (in this case, for data type S16) .....</i>	C3-539
<i>Figure C3-6</i>	<i>Operation of quadword VSHL.I64 Qd, Qm, #1 .....</i>	C3-572
<i>Figure C3-7</i>	<i>Operation of quadword VSLL.I64 Qd, Qm, #1 .....</i>	C3-577
<i>Figure C3-8</i>	<i>Operation of doubleword VSRI.I64 Dd, Dm, #2 .....</i>	C3-579
<i>Figure C3-9</i>	<i>Operation of doubleword VTRN.8 .....</i>	C3-592
<i>Figure C3-10</i>	<i>Operation of doubleword VTRN.32 .....</i>	C3-592



# List of Tables

## Arm® Instruction Set Reference Guide

<i>Table A2-1</i>	<i>AArch32 processor modes</i> .....	A2-55
<i>Table A2-2</i>	<i>Predeclared core registers in AArch32 state</i> .....	A2-61
<i>Table A2-3</i>	<i>Predeclared extension registers in AArch32 state</i> .....	A2-62
<i>Table A2-4</i>	<i>A32 instruction groups</i> .....	A2-68
<i>Table A3-1</i>	<i>Predeclared core registers in AArch64 state</i> .....	A3-76
<i>Table A3-2</i>	<i>Predeclared extension registers in AArch64 state</i> .....	A3-77
<i>Table A3-3</i>	<i>A64 instruction groups</i> .....	A3-83
<i>Table B1-1</i>	<i>Differences in syntax and mnemonics between A32/T32 and A64 Advanced SIMD instructions</i> .....	B1-95
<i>Table B1-2</i>	<i>Advanced SIMD data types</i> .....	B1-100
<i>Table B1-3</i>	<i>Advanced SIMD saturation ranges</i> .....	B1-104
<i>Table B2-1</i>	<i>Differences in syntax and mnemonics between A32/T32 and A64 floating-point instructions</i> .....	B2-120
<i>Table C1-1</i>	<i>Condition code suffixes</i> .....	C1-142
<i>Table C1-2</i>	<i>Condition code suffixes and related flags</i> .....	C1-143
<i>Table C1-3</i>	<i>Condition codes</i> .....	C1-144
<i>Table C1-4</i>	<i>Conditional branches only</i> .....	C1-147
<i>Table C1-5</i>	<i>All instructions conditional</i> .....	C1-148
<i>Table C2-1</i>	<i>Summary of instructions</i> .....	C2-156
<i>Table C2-2</i>	<i>PC-relative offsets</i> .....	C2-174
<i>Table C2-3</i>	<i>Register-relative offsets</i> .....	C2-176
<i>Table C2-4</i>	<i>B instruction availability and range</i> .....	C2-182
<i>Table C2-5</i>	<i>BL instruction availability and range</i> .....	C2-189

<i>Table C2-6</i>	<i>BLX instruction availability and range</i> .....	C2-190
<i>Table C2-7</i>	<i>BX instruction availability and range</i> .....	C2-192
<i>Table C2-8</i>	<i>BXJ instruction availability and range</i> .....	C2-194
<i>Table C2-9</i>	<i>Permitted instructions inside an IT block</i> .....	C2-223
<i>Table C2-10</i>	<i>Offsets and architectures, LDR, word, halfword, and byte</i> .....	C2-232
<i>Table C2-11</i>	<i>PC-relative offsets</i> .....	C2-234
<i>Table C2-12</i>	<i>Options and architectures, LDR (register offsets)</i> .....	C2-237
<i>Table C2-13</i>	<i>Register-relative offsets</i> .....	C2-238
<i>Table C2-14</i>	<i>Offsets and architectures, LDR (User mode)</i> .....	C2-240
<i>Table C2-15</i>	<i>Offsets and architectures, STR, word, halfword, and byte</i> .....	C2-359
<i>Table C2-16</i>	<i>Options and architectures, STR (register offsets)</i> .....	C2-361
<i>Table C2-17</i>	<i>Offsets and architectures, STR (User mode)</i> .....	C2-364
<i>Table C3-1</i>	<i>Summary of Advanced SIMD instructions</i> .....	C3-445
<i>Table C3-2</i>	<i>Summary of shared Advanced SIMD and floating-point instructions</i> .....	C3-448
<i>Table C3-3</i>	<i>Patterns for immediate value in VBIC (immediate)</i> .....	C3-462
<i>Table C3-4</i>	<i>Permitted combinations of parameters for VLDn (single n-element structure to one lane)</i> ....	C3-497
<i>Table C3-5</i>	<i>Permitted combinations of parameters for VLDn (single n-element structure to all lanes)</i> ....	C3-499
<i>Table C3-6</i>	<i>Permitted combinations of parameters for VLDn (multiple n-element structures)</i> .....	C3-501
<i>Table C3-7</i>	<i>Available immediate values in VMOV (immediate)</i> .....	C3-517
<i>Table C3-8</i>	<i>Available immediate values in VMVN (immediate)</i> .....	C3-531
<i>Table C3-9</i>	<i>Patterns for immediate value in VORR (immediate)</i> .....	C3-536
<i>Table C3-10</i>	<i>Available immediate ranges in VQRSHRN and VQRSHRUN (by immediate)</i> .....	C3-552
<i>Table C3-11</i>	<i>Available immediate ranges in VQSHL and VQSHLU (by immediate)</i> .....	C3-554
<i>Table C3-12</i>	<i>Available immediate ranges in VQSHRN and VQSHRUN (by immediate)</i> .....	C3-555
<i>Table C3-13</i>	<i>Results for out-of-range inputs in VRECPE</i> .....	C3-558
<i>Table C3-14</i>	<i>Results for out-of-range inputs in VRECPS</i> .....	C3-559
<i>Table C3-15</i>	<i>Available immediate ranges in VRSHR (by immediate)</i> .....	C3-563
<i>Table C3-16</i>	<i>Available immediate ranges in VRSHRN (by immediate)</i> .....	C3-564
<i>Table C3-17</i>	<i>Results for out-of-range inputs in VRSQRTE</i> .....	C3-566
<i>Table C3-18</i>	<i>Results for out-of-range inputs in VRSQRTS</i> .....	C3-567
<i>Table C3-19</i>	<i>Available immediate ranges in VRSRA (by immediate)</i> .....	C3-568
<i>Table C3-20</i>	<i>Available immediate ranges in VSHL (by immediate)</i> .....	C3-572
<i>Table C3-21</i>	<i>Available immediate ranges in VSHLL (by immediate)</i> .....	C3-574
<i>Table C3-22</i>	<i>Available immediate ranges in VSHR (by immediate)</i> .....	C3-575
<i>Table C3-23</i>	<i>Available immediate ranges in VSHRN (by immediate)</i> .....	C3-576
<i>Table C3-24</i>	<i>Available immediate ranges in VSRA (by immediate)</i> .....	C3-578
<i>Table C3-25</i>	<i>Permitted combinations of parameters for VSTn (multiple n-element structures)</i> .....	C3-581
<i>Table C3-26</i>	<i>Permitted combinations of parameters for VSTn (single n-element structure to one lane)</i> ....	C3-583
<i>Table C3-27</i>	<i>Operation of doubleword VUZP.8</i> .....	C3-596
<i>Table C3-28</i>	<i>Operation of quadword VUZP.32</i> .....	C3-596
<i>Table C3-29</i>	<i>Operation of doubleword VZIP.8</i> .....	C3-597
<i>Table C3-30</i>	<i>Operation of quadword VZIP.32</i> .....	C3-597
<i>Table C4-1</i>	<i>Summary of floating-point instructions</i> .....	C4-601
<i>Table C5-1</i>	<i>Summary of A32/T32 cryptographic instructions</i> .....	C5-644
<i>Table D1-1</i>	<i>Condition code suffixes</i> .....	D1-654
<i>Table D1-2</i>	<i>Condition code suffixes and related flags</i> .....	D1-655
<i>Table D2-1</i>	<i>Summary of A64 general instructions</i> .....	D2-662

<i>Table D2-2</i>	<i>ADD (64-bit general registers) specifier combinations</i> .....	D2-673
<i>Table D2-3</i>	<i>ADDS (64-bit general registers) specifier combinations</i> .....	D2-678
<i>Table D2-4</i>	<i>SYS parameter values corresponding to AT operations</i> .....	D2-690
<i>Table D2-5</i>	<i>CMN (64-bit general registers) specifier combinations</i> .....	D2-724
<i>Table D2-6</i>	<i>CMP (64-bit general registers) specifier combinations</i> .....	D2-728
<i>Table D2-7</i>	<i>SYS parameter values corresponding to DC operations</i> .....	D2-743
<i>Table D2-8</i>	<i>SYS parameter values corresponding to IC operations</i> .....	D2-763
<i>Table D2-9</i>	<i>SUB (64-bit general registers) specifier combinations</i> .....	D2-835
<i>Table D2-10</i>	<i>SUBS (64-bit general registers) specifier combinations</i> .....	D2-842
<i>Table D2-11</i>	<i>SYS parameter values corresponding to TLBI operations</i> .....	D2-853
<i>Table D3-1</i>	<i>Summary of A64 data transfer instructions</i> .....	D3-877
<i>Table D4-1</i>	<i>Summary of A64 floating-point instructions</i> .....	D4-1029
<i>Table D5-1</i>	<i>Summary of A64 SIMD scalar instructions</i> .....	D5-1110
<i>Table D5-2</i>	<i>DUP (Scalar) specifier combinations</i> .....	D5-1129
<i>Table D5-3</i>	<i>FCMLA (Scalar) specifier combinations</i> .....	D5-1141
<i>Table D5-4</i>	<i>FCVTZS (Scalar) specifier combinations</i> .....	D5-1153
<i>Table D5-5</i>	<i>FCVTZU (Scalar) specifier combinations</i> .....	D5-1155
<i>Table D5-6</i>	<i>FMLA (Scalar, single-precision and double-precision) specifier combinations</i> .....	D5-1162
<i>Table D5-7</i>	<i>FMLS (Scalar, single-precision and double-precision) specifier combinations</i> .....	D5-1165
<i>Table D5-8</i>	<i>FMUL (Scalar, single-precision and double-precision) specifier combinations</i> .....	D5-1168
<i>Table D5-9</i>	<i>FMULX (Scalar, single-precision and double-precision) specifier combinations</i> .....	D5-1170
<i>Table D5-10</i>	<i>MOV (Scalar) specifier combinations</i> .....	D5-1176
<i>Table D5-11</i>	<i>SCVTF (Scalar) specifier combinations</i> .....	D5-1178
<i>Table D5-12</i>	<i>SQDMLAL (Scalar) specifier combinations</i> .....	D5-1184
<i>Table D5-13</i>	<i>SQDMLAL (Scalar) specifier combinations</i> .....	D5-1185
<i>Table D5-14</i>	<i>SQDMLSL (Scalar) specifier combinations</i> .....	D5-1186
<i>Table D5-15</i>	<i>SQDMLSL (Scalar) specifier combinations</i> .....	D5-1187
<i>Table D5-16</i>	<i>SQDMULH (Scalar) specifier combinations</i> .....	D5-1188
<i>Table D5-17</i>	<i>SQDMULL (Scalar) specifier combinations</i> .....	D5-1190
<i>Table D5-18</i>	<i>SQDMULL (Scalar) specifier combinations</i> .....	D5-1191
<i>Table D5-19</i>	<i>SQRDMLAH (Scalar) specifier combinations</i> .....	D5-1193
<i>Table D5-20</i>	<i>SQRDMLSH (Scalar) specifier combinations</i> .....	D5-1195
<i>Table D5-21</i>	<i>SQRDMULH (Scalar) specifier combinations</i> .....	D5-1197
<i>Table D5-22</i>	<i>SQRSHRN (Scalar) specifier combinations</i> .....	D5-1200
<i>Table D5-23</i>	<i>SQRSHRUN (Scalar) specifier combinations</i> .....	D5-1201
<i>Table D5-24</i>	<i>SQSHL (Scalar) specifier combinations</i> .....	D5-1202
<i>Table D5-25</i>	<i>SQSHLU (Scalar) specifier combinations</i> .....	D5-1204
<i>Table D5-26</i>	<i>SQSHRN (Scalar) specifier combinations</i> .....	D5-1205
<i>Table D5-27</i>	<i>SQSHRUN (Scalar) specifier combinations</i> .....	D5-1206
<i>Table D5-28</i>	<i>SQXTN (Scalar) specifier combinations</i> .....	D5-1208
<i>Table D5-29</i>	<i>SQXTUN (Scalar) specifier combinations</i> .....	D5-1209
<i>Table D5-30</i>	<i>UCVTF (Scalar) specifier combinations</i> .....	D5-1219
<i>Table D5-31</i>	<i>UQRSHRN (Scalar) specifier combinations</i> .....	D5-1223
<i>Table D5-32</i>	<i>UQSHL (Scalar) specifier combinations</i> .....	D5-1224
<i>Table D5-33</i>	<i>UQSHRN (Scalar) specifier combinations</i> .....	D5-1226
<i>Table D5-34</i>	<i>UQXTN (Scalar) specifier combinations</i> .....	D5-1228
<i>Table D6-1</i>	<i>Summary of A64 SIMD Vector instructions</i> .....	D6-1243
<i>Table D6-2</i>	<i>ADDHN, ADDHN2 (Vector) specifier combinations</i> .....	D6-1256
<i>Table D6-3</i>	<i>ADDV (Vector) specifier combinations</i> .....	D6-1258
<i>Table D6-4</i>	<i>DUP (Vector) specifier combinations</i> .....	D6-1279

<i>Table D6-5</i>	<i>DUP (Vector) specifier combinations</i> .....	D6-1280
<i>Table D6-6</i>	<i>EXT (Vector) specifier combinations</i> .....	D6-1282
<i>Table D6-7</i>	<i>FCVTL, FCVTL2 (Vector) specifier combinations</i> .....	D6-1301
<i>Table D6-8</i>	<i>FCVTN, FCVTN2 (Vector) specifier combinations</i> .....	D6-1304
<i>Table D6-9</i>	<i>FCVTXN{2} (Vector) specifier combinations</i> .....	D6-1309
<i>Table D6-10</i>	<i>FCVTZS (Vector) specifier combinations</i> .....	D6-1310
<i>Table D6-11</i>	<i>FCVTZU (Vector) specifier combinations</i> .....	D6-1312
<i>Table D6-12</i>	<i>FMLA (Vector, single-precision and double-precision) specifier combinations</i> .....	D6-1328
<i>Table D6-13</i>	<i>FMLS (Vector, single-precision and double-precision) specifier combinations</i> .....	D6-1332
<i>Table D6-14</i>	<i>FMUL (Vector, single-precision and double-precision) specifier combinations</i> .....	D6-1338
<i>Table D6-15</i>	<i>FMULX (Vector, single-precision and double-precision) specifier combinations</i> .....	D6-1341
<i>Table D6-16</i>	<i>INS (Vector) specifier combinations</i> .....	D6-1358
<i>Table D6-17</i>	<i>INS (Vector) specifier combinations</i> .....	D6-1359
<i>Table D6-18</i>	<i>LD1 (One register, immediate offset) specifier combinations</i> .....	D6-1361
<i>Table D6-19</i>	<i>LD1 (Two registers, immediate offset) specifier combinations</i> .....	D6-1361
<i>Table D6-20</i>	<i>LD1 (Three registers, immediate offset) specifier combinations</i> .....	D6-1361
<i>Table D6-21</i>	<i>LD1 (Four registers, immediate offset) specifier combinations</i> .....	D6-1362
<i>Table D6-22</i>	<i>LD1R (Immediate offset) specifier combinations</i> .....	D6-1364
<i>Table D6-23</i>	<i>LD2R (Immediate offset) specifier combinations</i> .....	D6-1367
<i>Table D6-24</i>	<i>LD3R (Immediate offset) specifier combinations</i> .....	D6-1371
<i>Table D6-25</i>	<i>LD4R (Immediate offset) specifier combinations</i> .....	D6-1375
<i>Table D6-26</i>	<i>MLA (Vector) specifier combinations</i> .....	D6-1376
<i>Table D6-27</i>	<i>MLS (Vector) specifier combinations</i> .....	D6-1378
<i>Table D6-28</i>	<i>MOV (Vector) specifier combinations</i> .....	D6-1380
<i>Table D6-29</i>	<i>MOV (Vector) specifier combinations</i> .....	D6-1381
<i>Table D6-30</i>	<i>MUL (Vector) specifier combinations</i> .....	D6-1386
<i>Table D6-31</i>	<i>PMULL, PMULL2 (Vector) specifier combinations</i> .....	D6-1396
<i>Table D6-32</i>	<i>RADDHN, RADDHN2 (Vector) specifier combinations</i> .....	D6-1397
<i>Table D6-33</i>	<i>RSHRN, RSHRN2 (Vector) specifier combinations</i> .....	D6-1402
<i>Table D6-34</i>	<i>RSUBHN, RSUBHN2 (Vector) specifier combinations</i> .....	D6-1403
<i>Table D6-35</i>	<i>SABAL, SABAL2 (Vector) specifier combinations</i> .....	D6-1405
<i>Table D6-36</i>	<i>SABDL, SABDL2 (Vector) specifier combinations</i> .....	D6-1407
<i>Table D6-37</i>	<i>SADALP (Vector) specifier combinations</i> .....	D6-1408
<i>Table D6-38</i>	<i>SADDL, SADDL2 (Vector) specifier combinations</i> .....	D6-1409
<i>Table D6-39</i>	<i>SADDLP (Vector) specifier combinations</i> .....	D6-1410
<i>Table D6-40</i>	<i>SADDLV (Vector) specifier combinations</i> .....	D6-1411
<i>Table D6-41</i>	<i>SADDW, SADDW2 (Vector) specifier combinations</i> .....	D6-1412
<i>Table D6-42</i>	<i>SCVTF (Vector) specifier combinations</i> .....	D6-1413
<i>Table D6-43</i>	<i>SHL (Vector) specifier combinations</i> .....	D6-1418
<i>Table D6-44</i>	<i>SHLL, SHLL2 (Vector) specifier combinations</i> .....	D6-1419
<i>Table D6-45</i>	<i>SHRN, SHRN2 (Vector) specifier combinations</i> .....	D6-1420
<i>Table D6-46</i>	<i>SLI (Vector) specifier combinations</i> .....	D6-1422
<i>Table D6-47</i>	<i>SMAXV (Vector) specifier combinations</i> .....	D6-1425
<i>Table D6-48</i>	<i>SMINV (Vector) specifier combinations</i> .....	D6-1428
<i>Table D6-49</i>	<i>SMLAL, SMLAL2 (Vector) specifier combinations</i> .....	D6-1429
<i>Table D6-50</i>	<i>SMLAL, SMLAL2 (Vector) specifier combinations</i> .....	D6-1430
<i>Table D6-51</i>	<i>SMLSL, SMLSL2 (Vector) specifier combinations</i> .....	D6-1431
<i>Table D6-52</i>	<i>SMLSL, SMLSL2 (Vector) specifier combinations</i> .....	D6-1432
<i>Table D6-53</i>	<i>SMOV (32-bit) specifier combinations</i> .....	D6-1433
<i>Table D6-54</i>	<i>SMOV (64-bit) specifier combinations</i> .....	D6-1433

<a href="#">Table D6-55</a>	<a href="#">SMULL, SMULL2 (Vector) specifier combinations</a>	<a href="#">D6-1434</a>
<a href="#">Table D6-56</a>	<a href="#">SMULL, SMULL2 (Vector) specifier combinations</a>	<a href="#">D6-1435</a>
<a href="#">Table D6-57</a>	<a href="#">SQDMLAL{2} (Vector) specifier combinations</a>	<a href="#">D6-1438</a>
<a href="#">Table D6-58</a>	<a href="#">SQDMLAL{2} (Vector) specifier combinations</a>	<a href="#">D6-1440</a>
<a href="#">Table D6-59</a>	<a href="#">SQDMLSL{2} (Vector) specifier combinations</a>	<a href="#">D6-1441</a>
<a href="#">Table D6-60</a>	<a href="#">SQDMLSL{2} (Vector) specifier combinations</a>	<a href="#">D6-1443</a>
<a href="#">Table D6-61</a>	<a href="#">SQDMULH (Vector) specifier combinations</a>	<a href="#">D6-1444</a>
<a href="#">Table D6-62</a>	<a href="#">SQDMULL{2} (Vector) specifier combinations</a>	<a href="#">D6-1446</a>
<a href="#">Table D6-63</a>	<a href="#">SQDMULL{2} (Vector) specifier combinations</a>	<a href="#">D6-1448</a>
<a href="#">Table D6-64</a>	<a href="#">SQRDMLAH (Vector) specifier combinations</a>	<a href="#">D6-1450</a>
<a href="#">Table D6-65</a>	<a href="#">SQRDMLSH (Vector) specifier combinations</a>	<a href="#">D6-1452</a>
<a href="#">Table D6-66</a>	<a href="#">SQRDMULH (Vector) specifier combinations</a>	<a href="#">D6-1454</a>
<a href="#">Table D6-67</a>	<a href="#">SQRSHRN{2} (Vector) specifier combinations</a>	<a href="#">D6-1457</a>
<a href="#">Table D6-68</a>	<a href="#">SQRSHRUN{2} (Vector) specifier combinations</a>	<a href="#">D6-1458</a>
<a href="#">Table D6-69</a>	<a href="#">SQSHL (Vector) specifier combinations</a>	<a href="#">D6-1459</a>
<a href="#">Table D6-70</a>	<a href="#">SQSHLU (Vector) specifier combinations</a>	<a href="#">D6-1461</a>
<a href="#">Table D6-71</a>	<a href="#">SQSHRN{2} (Vector) specifier combinations</a>	<a href="#">D6-1462</a>
<a href="#">Table D6-72</a>	<a href="#">SQSHRUN{2} (Vector) specifier combinations</a>	<a href="#">D6-1463</a>
<a href="#">Table D6-73</a>	<a href="#">SQXTN{2} (Vector) specifier combinations</a>	<a href="#">D6-1465</a>
<a href="#">Table D6-74</a>	<a href="#">SQXTUN{2} (Vector) specifier combinations</a>	<a href="#">D6-1466</a>
<a href="#">Table D6-75</a>	<a href="#">SRI (Vector) specifier combinations</a>	<a href="#">D6-1468</a>
<a href="#">Table D6-76</a>	<a href="#">SRSHR (Vector) specifier combinations</a>	<a href="#">D6-1470</a>
<a href="#">Table D6-77</a>	<a href="#">SRSRA (Vector) specifier combinations</a>	<a href="#">D6-1471</a>
<a href="#">Table D6-78</a>	<a href="#">SSHLL, SSHLL2 (Vector) specifier combinations</a>	<a href="#">D6-1473</a>
<a href="#">Table D6-79</a>	<a href="#">SSHR (Vector) specifier combinations</a>	<a href="#">D6-1474</a>
<a href="#">Table D6-80</a>	<a href="#">SSRA (Vector) specifier combinations</a>	<a href="#">D6-1475</a>
<a href="#">Table D6-81</a>	<a href="#">SSUBL, SSUBL2 (Vector) specifier combinations</a>	<a href="#">D6-1476</a>
<a href="#">Table D6-82</a>	<a href="#">SSUBW, SSUBW2 (Vector) specifier combinations</a>	<a href="#">D6-1477</a>
<a href="#">Table D6-83</a>	<a href="#">ST1 (One register, immediate offset) specifier combinations</a>	<a href="#">D6-1479</a>
<a href="#">Table D6-84</a>	<a href="#">ST1 (Two registers, immediate offset) specifier combinations</a>	<a href="#">D6-1479</a>
<a href="#">Table D6-85</a>	<a href="#">ST1 (Three registers, immediate offset) specifier combinations</a>	<a href="#">D6-1479</a>
<a href="#">Table D6-86</a>	<a href="#">ST1 (Four registers, immediate offset) specifier combinations</a>	<a href="#">D6-1480</a>
<a href="#">Table D6-87</a>	<a href="#">SUBHN, SUBHN2 (Vector) specifier combinations</a>	<a href="#">D6-1491</a>
<a href="#">Table D6-88</a>	<a href="#">SXTL, SXTL2 (Vector) specifier combinations</a>	<a href="#">D6-1493</a>
<a href="#">Table D6-89</a>	<a href="#">UABAL, UABAL2 (Vector) specifier combinations</a>	<a href="#">D6-1499</a>
<a href="#">Table D6-90</a>	<a href="#">UABDL, UABDL2 (Vector) specifier combinations</a>	<a href="#">D6-1501</a>
<a href="#">Table D6-91</a>	<a href="#">UADALP (Vector) specifier combinations</a>	<a href="#">D6-1502</a>
<a href="#">Table D6-92</a>	<a href="#">UADDL, UADDL2 (Vector) specifier combinations</a>	<a href="#">D6-1503</a>
<a href="#">Table D6-93</a>	<a href="#">UADDLP (Vector) specifier combinations</a>	<a href="#">D6-1504</a>
<a href="#">Table D6-94</a>	<a href="#">UADDLV (Vector) specifier combinations</a>	<a href="#">D6-1505</a>
<a href="#">Table D6-95</a>	<a href="#">UADDW, UADDW2 (Vector) specifier combinations</a>	<a href="#">D6-1506</a>
<a href="#">Table D6-96</a>	<a href="#">UCVTF (Vector) specifier combinations</a>	<a href="#">D6-1507</a>
<a href="#">Table D6-97</a>	<a href="#">UMAXV (Vector) specifier combinations</a>	<a href="#">D6-1515</a>
<a href="#">Table D6-98</a>	<a href="#">UMINV (Vector) specifier combinations</a>	<a href="#">D6-1518</a>
<a href="#">Table D6-99</a>	<a href="#">UMLAL, UMLAL2 (Vector) specifier combinations</a>	<a href="#">D6-1519</a>
<a href="#">Table D6-100</a>	<a href="#">UMLAL, UMLAL2 (Vector) specifier combinations</a>	<a href="#">D6-1520</a>
<a href="#">Table D6-101</a>	<a href="#">UMLSL, UMLSL2 (Vector) specifier combinations</a>	<a href="#">D6-1521</a>
<a href="#">Table D6-102</a>	<a href="#">UMLSL, UMLSL2 (Vector) specifier combinations</a>	<a href="#">D6-1522</a>
<a href="#">Table D6-103</a>	<a href="#">UMOV (32-bit) specifier combinations</a>	<a href="#">D6-1523</a>
<a href="#">Table D6-104</a>	<a href="#">UMULL, UMULL2 (Vector) specifier combinations</a>	<a href="#">D6-1524</a>

<i>Table D6-105</i>	UMULL, UMULL2 (Vector) specifier combinations .....	D6-1525
<i>Table D6-106</i>	UQRSHRN{2} (Vector) specifier combinations .....	D6-1528
<i>Table D6-107</i>	UQSHL (Vector) specifier combinations .....	D6-1529
<i>Table D6-108</i>	UQSHRN{2} (Vector) specifier combinations .....	D6-1531
<i>Table D6-109</i>	UQXTN{2} (Vector) specifier combinations .....	D6-1534
<i>Table D6-110</i>	URSHR (Vector) specifier combinations .....	D6-1538
<i>Table D6-111</i>	URSRA (Vector) specifier combinations .....	D6-1540
<i>Table D6-112</i>	USHLL, USHLL2 (Vector) specifier combinations .....	D6-1542
<i>Table D6-113</i>	USHR (Vector) specifier combinations .....	D6-1543
<i>Table D6-114</i>	USRA (Vector) specifier combinations .....	D6-1545
<i>Table D6-115</i>	USUBL, USUBL2 (Vector) specifier combinations .....	D6-1546
<i>Table D6-116</i>	USUBW, USUBW2 (Vector) specifier combinations .....	D6-1547
<i>Table D6-117</i>	UXTL, UXTL2 (Vector) specifier combinations .....	D6-1548
<i>Table D6-118</i>	XTN, XTN2 (Vector) specifier combinations .....	D6-1551
<i>Table D7-1</i>	Summary of A64 cryptographic instructions .....	D7-1556

# Preface

This preface introduces the *Arm® Instruction Set Reference Guide*.

It contains the following:

- [About this book](#) on page 42.

## About this book

Arm® Instruction Set Reference Guide. This document contains an overview of the Arm architecture and information on A32, T32, and A64 instruction sets. For assembler-specific features, such as additional pseudo-instructions, see the documentation for your assembler.

## Using this book

This book is organized into the following chapters:

### **Part A Instruction Set Overview**

#### **Chapter A1 Overview of the Arm® Architecture**

Gives an overview of the Arm architecture.

#### **Chapter A2 Overview of AArch32 state**

Gives an overview of the AArch32 state of Armv8.

#### **Chapter A3 Overview of AArch64 state**

Gives an overview of the AArch64 state of Armv8.

### **Part B Advanced SIMD and Floating-point Programming**

#### **Chapter B1 Advanced SIMD Programming**

Describes Advanced SIMD assembly language programming.

#### **Chapter B2 Floating-point Programming**

Describes floating-point assembly language programming.

### **Part C A32/T32 Instruction Set Reference**

#### **Chapter C1 Condition Codes**

Describes condition codes and conditional execution of A32 and T32 code.

#### **Chapter C2 A32 and T32 Instructions**

Describes the A32 and T32 instructions supported in AArch32 state.

#### **Chapter C3 Advanced SIMD Instructions (32-bit)**

Describes Advanced SIMD assembly language instructions.

#### **Chapter C4 Floating-point Instructions (32-bit)**

Describes floating-point assembly language instructions.

#### **Chapter C5 A32/T32 Cryptographic Algorithms**

Lists the algorithms that A32 and T32 SIMD instructions support.

### **Part D A64 Instruction Set Reference**

#### **Chapter D1 Condition Codes**

Describes condition codes and conditional execution of A64 code.

#### **Chapter D2 A64 General Instructions**

Describes the A64 general instructions.

#### **Chapter D3 A64 Data Transfer Instructions**

Describes the A64 data transfer instructions.

#### **Chapter D4 A64 Floating-point Instructions**

Describes the A64 floating-point instructions.

#### **Chapter D5 A64 SIMD Scalar Instructions**

Describes the A64 SIMD scalar instructions.

## Chapter D6 A64 SIMD Vector Instructions

Describes the A64 SIMD vector instructions.

## Chapter D7 A64 Cryptographic Algorithms

Lists the algorithms that A64 SIMD instructions support.

## Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the [Arm® Glossary](#) for more information.

## Typographic conventions

### *italic*

Introduces special terminology, denotes cross-references, and citations.

### **bold**

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

### `monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

### monospace

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

### `monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

### `monospace bold`

Denotes language keywords when used outside example code.

### `<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

### SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the [Arm® Glossary](#). For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

## Feedback

### Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

### Feedback on content

If you have comments on content then send an e-mail to [errata@arm.com](mailto:errata@arm.com). Give:

- The title *Arm Instruction Set Reference Guide*.
- The number 100076\_0100\_00\_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

———— Note ————

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

## Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

# **Part A**

## **Instruction Set Overview**



# Chapter A1

## Overview of the Arm® Architecture

Gives an overview of the Arm architecture.

It contains the following sections:

- [\*A1.1 About the Arm® architecture\* on page A1-48.](#)
- [\*A1.2 Differences between the A64, A32, and T32 instruction sets\* on page A1-49.](#)
- [\*A1.3 Changing between AArch64 and AArch32 states\* on page A1-50.](#)
- [\*A1.4 Advanced SIMD\* on page A1-51.](#)
- [\*A1.5 Floating-point hardware\* on page A1-52.](#)

## A1.1 About the Arm® architecture

The Arm architecture is a load-store architecture. The addressing range depends on whether you are using the 32-bit or the 64-bit architecture.

Arm processors are typical of RISC processors in that only load and store instructions can access memory. Data processing instructions operate on register contents only.

Armv8 is the next major architectural update after Armv7. It introduces a 64-bit architecture, but maintains compatibility with existing 32-bit architectures. It uses two execution states:

### AArch32

In AArch32 state, code has access to 32-bit general purpose registers.

Code executing in AArch32 state can only use the A32 and T32 instruction sets. This state is broadly compatible with the Armv7-A architecture.

### AArch64

In AArch64 state, code has access to 64-bit general purpose registers. The AArch64 state exists only in the Armv8 architecture.

Code executing in AArch64 state can only use the A64 instruction set.

#### *Related information*

*Arm Architecture Reference Manual*

## A1.2 Differences between the A64, A32, and T32 instruction sets

A64 instructions are 32 bits wide.

Armv8 introduces a new set of 32-bit instructions called A64, with new encodings and assembly language. A64 is only available when the processor is in AArch64 state. It provides similar functionality to the A32 and T32 instruction sets, but gives access to a larger virtual address space, and has some other changes, including reduced conditionality.

Armv8 also defines an optional Crypto Extension. This extension provides cryptographic and hash instructions in the A64 instruction set.

### ***Related reference***

[A3.12 A64 instruction set overview on page A3-83](#)

## A1.3 Changing between AArch64 and AArch32 states

The processor must be in the correct execution state for the instructions it is executing.

A processor that is executing A64 instructions is operating in AArch64 state. In this state, the instructions can access both the 64-bit and 32-bit registers.

A processor that is executing A32 or T32 instructions is operating in AArch32 state. In this state, the instructions can only access the 32-bit registers, and not the 64-bit registers.

A processor based on the Armv8 architecture can run applications built for AArch32 and AArch64 states but a change between AArch32 and AArch64 states can only happen at exception boundaries.

Arm Compiler toolchain builds images for either the AArch32 state or AArch64 state. Therefore, an image built with Arm Compiler toolchain can either contain only A32 and T32 instructions or only A64 instructions.

A processor can only execute instructions from the instruction set that matches its current execution state. A processor in AArch32 state cannot execute A64 instructions, and a processor in AArch64 state cannot execute A32 or T32 instructions. You must ensure that the processor never receives instructions from the wrong instruction set for the current execution state.

### ***Related reference***

[C2.20 BLX, BLXNS on page C2-190](#)

[C2.21 BX, BXNS on page C2-192](#)

## A1.4 Advanced SIMD

Advanced SIMD is a 64-bit and 128-bit hybrid *Single Instruction Multiple Data* (SIMD) technology targeted at advanced media and signal processing applications and embedded processors.

Advanced SIMD is implemented as part of an Arm-based processor, but has its own execution pipelines and a register bank that is distinct from the general-purpose register bank.

Advanced SIMD instructions are available in both A32 and A64. The A64 Advanced SIMD instructions are based on those in A32. The main differences are the following:

- Different instruction mnemonics and syntax.
- Thirty-two 128-bit vector registers, increased from sixteen in A32.
- A different register packing scheme:
  - In A64, smaller registers occupy the low order bits of larger registers. For example, S31 maps to bits[31:0] of D31.
  - In A32, smaller registers are packed into larger registers. For example, S31 maps to bits[63:32] of D15.
- A64 Advanced SIMD instructions support both single-precision and double-precision floating-point data types and arithmetic.
- A32 Advanced SIMD instructions support only single-precision floating-point data types.

### ***Related reference***

*Chapter B1 Advanced SIMD Programming* on page B1-87

## A1.5 Floating-point hardware

There are several floating-point architecture versions and variants.

The floating-point hardware, together with associated support code, provides single-precision and double-precision floating-point arithmetic, as defined by *IEEE Std. 754-2008 IEEE Standard for Floating-Point Arithmetic*. This document is referred to as the IEEE 754 standard.

The floating-point hardware uses a register bank that is distinct from the Arm core register bank.

————— **Note** —————

The floating-point register bank is shared with the SIMD register bank.

In AArch32 state, floating-point support is largely unchanged from VFPv4, apart from the addition of a few instructions for compliance with the IEEE 754 standard.

The floating-point architecture in AArch64 state is also based on VFPv4. The main differences are the following:

- In AArch64 state, the number of 128-bit SIMD and floating-point registers increases from sixteen to thirty-two.
- Single-precision registers are no longer packed into double-precision registers, so register Sx is Dx[31:0].
- The presence of floating-point hardware is mandated, so software floating-point linkage is not supported.
- Earlier versions of the floating-point architecture, for instance VFPv2, VFPv3, and VFPv4, are not supported in AArch64 state.
- VFP vector mode is not supported in either AArch32 or AArch64 state. Use Advanced SIMD instructions for vector floating-point.
- Some new instructions have been added, including:
  - Direct conversion between half-precision and double-precision.
  - Load and store pair, replacing load and store multiple.
  - Fused multiply-add and multiply-subtract.
  - Instructions for IEEE 754-2008 compatibility.

**Related concepts**

[B2.5 Views of the floating-point extension register bank in AArch64 state](#) on page B2-119

# Chapter A2

## Overview of AArch32 state

Gives an overview of the AArch32 state of Armv8.

It contains the following sections:

- [\*A2.1 Changing between A32 and T32 instruction set states\* on page A2-54.](#)
- [\*A2.2 Processor modes, and privileged and unprivileged software execution\* on page A2-55.](#)
- [\*A2.3 Processor modes in Armv6-M, Armv7-M, and Armv8-M\* on page A2-56.](#)
- [\*A2.4 Registers in AArch32 state\* on page A2-57.](#)
- [\*A2.5 General-purpose registers in AArch32 state\* on page A2-59.](#)
- [\*A2.6 Register accesses in AArch32 state\* on page A2-60.](#)
- [\*A2.7 Predeclared core register names in AArch32 state\* on page A2-61.](#)
- [\*A2.8 Predeclared extension register names in AArch32 state\* on page A2-62.](#)
- [\*A2.9 Program Counter in AArch32 state\* on page A2-63.](#)
- [\*A2.10 The Q flag in AArch32 state\* on page A2-64.](#)
- [\*A2.11 Application Program Status Register\* on page A2-65.](#)
- [\*A2.12 Current Program Status Register in AArch32 state\* on page A2-66.](#)
- [\*A2.13 Saved Program Status Registers in AArch32 state\* on page A2-67.](#)
- [\*A2.14 A32 and T32 instruction set overview\* on page A2-68.](#)
- [\*A2.15 Access to the inline barrel shifter in AArch32 state\* on page A2-69.](#)

## A2.1 Changing between A32 and T32 instruction set states

A processor that is executing A32 instructions is operating in *A32 instruction set state*. A processor that is executing T32 instructions is operating in *T32 instruction set state*. For brevity, this document refers to them as the *A32 state* and *T32 state* respectively.

A processor in A32 state cannot execute T32 instructions, and a processor in T32 state cannot execute A32 instructions. You must ensure that the processor never receives instructions of the wrong instruction set for the current state.

The initial state after reset depends on the processor being used and its configuration.

To direct `armasm` to generate A32 or T32 instruction encodings, you must set the assembler mode using an ARM or THUMB directive. Assembly code using CODE32 and CODE16 directives can still be assembled, but Arm recommends you use the ARM and THUMB directives for new code.

These directives do not change the instruction set state of the processor. To do this, you must use an appropriate instruction, for example BX or BLX to change between A32 and T32 states when performing a branch.

### *Related reference*

[C2.20 BLX, BLXNS on page C2-190](#)

[C2.21 BX, BXNS on page C2-192](#)

## A2.2 Processor modes, and privileged and unprivileged software execution

The Arm architecture supports different levels of execution privilege. The privilege level depends on the processor mode.

————— Note ————

Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline do not support the same modes as other Arm architectures and profiles. Some of the processor modes listed here do not apply to these architectures.

**Table A2-1 AArch32 processor modes**

Processor mode	Mode number
User	0b10000
FIQ	0b10001
IRQ	0b10010
Supervisor	0b10011
Monitor	0b10110
Abort	0b10111
Hyp	0b11010
Undefined	0b11011
System	0b11111

User mode is an unprivileged mode, and has restricted access to system resources. All other modes have full access to system resources in the current security state, can change mode freely, and execute software as privileged.

Applications that require task protection usually execute in User mode. Some embedded applications might run entirely in any mode other than User mode. An application that requires full access to system resources usually executes in System mode.

Modes other than User mode are entered to service exceptions, or to access privileged resources.

Code can run in either a Secure state or in a Non-secure state. Hypervisor (Hyp) mode has privileged execution in Non-secure state.

**Related concepts**

[A2.3 Processor modes in Armv6-M, Armv7-M, and Armv8-M on page A2-56](#)

**Related information**

[Arm Architecture Reference Manual](#)

## A2.3 Processor modes in Armv6-M, Armv7-M, and Armv8-M

The processor modes available in Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline are Thread mode and Handler mode.

Thread mode is the normal mode that programs run in. Thread mode can be privileged or unprivileged software execution. Handler mode is the mode that exceptions are handled in. It is always privileged software execution.

### ***Related concepts***

[A2.2 Processor modes, and privileged and unprivileged software execution](#) on page A2-55

### ***Related information***

[Arm Architecture Reference Manual](#)

## A2.4 Registers in AArch32 state

Arm processors provide general-purpose and special-purpose registers. Some additional registers are available in privileged execution modes.

In all Arm processors in AArch32 state, the following registers are available and accessible in any processor mode:

- 15 general-purpose registers R0-R12, the *Stack Pointer* (SP), and *Link Register* (LR).
- 1 *Program Counter* (PC).
- 1 *Application Program Status Register* (APSR).

————— Note ————

- SP and LR can be used as general-purpose registers, although Arm deprecates using SP other than as a stack pointer.

Additional registers are available in privileged software execution. Arm processors have a total of 43 registers. The registers are arranged in partially overlapping banks. There is a different register bank for each processor mode. The banked registers give rapid context switching for dealing with processor exceptions and privileged operations.

The additional registers in Arm processors are:

- 2 supervisor mode registers for banked SP and LR.
- 2 abort mode registers for banked SP and LR.
- 2 undefined mode registers for banked SP and LR.
- 2 interrupt mode registers for banked SP and LR.
- 7 FIQ mode registers for banked R8-R12, SP and LR.
- 2 monitor mode registers for banked SP and LR.
- 1 Hyp mode register for banked SP.
- 7 *Saved Program Status Register* (SPSRs), one for each exception mode.
- 1 Hyp mode register for ELR\_Hyp to store the preferred return address from Hyp mode.

————— Note ————

In privileged software execution, CPSR is an alias for APSR and gives access to additional bits.

The following figure shows how the registers are banked in the Arm architecture.

		User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr									
R1	R1_usr									
R2	R2_usr									
R3	R3_usr									
R4	R4_usr									
R5	R5_usr									
R6	R6_usr									
R7	R7_usr									
R8	R8_usr								R8_fiq	
R9	R9_usr								R9_fiq	
R10	R10_usr								R10_fiq	
R11	R11_usr								R11_fiq	
R12	R12_usr								R12_fiq	
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq	
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq	
PC	PC									
APSR	CPSR									
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq	
			ELR_hyp							

‡ Exists only in Secure state.

† Exists only in Non-secure state.

Cells with no entry indicate that the User mode register is used.

**Figure A2-1 Organization of general-purpose registers and Program Status Registers**

In Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline based processors, SP is an alias for the two banked stack pointer registers:

- Main stack pointer register, that is only available in privileged software execution.
- Process stack pointer register.

#### Related concepts

[A2.5 General-purpose registers in AArch32 state on page A2-59](#)

[A2.9 Program Counter in AArch32 state on page A2-63](#)

[A2.11 Application Program Status Register on page A2-65](#)

[A2.13 Saved Program Status Registers in AArch32 state on page A2-67](#)

[A2.12 Current Program Status Register in AArch32 state on page A2-66](#)

[A2.2 Processor modes, and privileged and unprivileged software execution on page A2-55](#)

#### Related information

[Arm Architecture Reference Manual](#)

## A2.5 General-purpose registers in AArch32 state

There are restrictions on the use of SP and LR as general-purpose registers.

With the exception of Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline based processors, there are 33 general-purpose 32-bit registers, including the banked SP and LR registers. Fifteen general-purpose registers are visible at any one time, depending on the current processor mode. These are R0-R12, SP, and LR. The PC (R15) is not considered a general-purpose register.

SP (or R13) is the *stack pointer*. The C and C++ compilers always use SP as the stack pointer. Arm deprecates most uses of SP as a general purpose register. In T32 state, SP is strictly defined as the stack pointer. The instruction descriptions in [Chapter C2 A32 and T32 Instructions](#) on page C2-151 describe when SP and PC can be used.

In User mode, LR (or R14) is used as a *link register* to store the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored on the stack.

In the exception handling modes, LR holds the return address for the exception, or a subroutine return address if subroutine calls are executed within an exception. LR can be used as a general-purpose register if the return address is stored on the stack.

### **Related concepts**

[A2.9 Program Counter in AArch32 state](#) on page A2-63

[A2.6 Register accesses in AArch32 state](#) on page A2-60

### **Related reference**

[A2.7 Predeclared core register names in AArch32 state](#) on page A2-61

[C2.65 MRS \(PSR to general-purpose register\)](#) on page C2-257

[C2.68 MSR \(general-purpose register to PSR\)](#) on page C2-261

## A2.6 Register accesses in AArch32 state

16-bit T32 instructions can access only a limited set of registers. There are also some restrictions on the use of special-purpose registers by A32 and 32-bit T32 instructions.

Most 16-bit T32 instructions can only access R0 to R7. Only a small number of T32 instructions can access R8-R12, SP, LR, and PC. Registers R0 to R7 are called Lo registers. Registers R8-R12, SP, LR, and PC are called Hi registers.

All 32-bit T32 instructions can access R0 to R12, and LR. However, apart from a few designated stack manipulation instructions, most T32 instructions cannot use SP. Except for a few specific instructions where PC is useful, most T32 instructions cannot use PC.

In A32 state, all instructions can access R0 to R12, SP, and LR, and most instructions can also access PC (R15). However, the use of the SP in an A32 instruction, in any way that is not possible in the corresponding T32 instruction, is deprecated. Explicit use of the PC in an A32 instruction is not usually useful, and except for specific instances that are useful, such use is deprecated. Implicit use of the PC, for example in branch instructions or load (literal) instructions, is never deprecated.

The MRS instructions can move the contents of a status register to a general-purpose register, where they can be manipulated by normal data processing operations. You can use the MSR instruction to move the contents of a general-purpose register to a status register.

### Related concepts

[A2.5 General-purpose registers in AArch32 state on page A2-59](#)

[A2.9 Program Counter in AArch32 state on page A2-63](#)

[A2.11 Application Program Status Register on page A2-65](#)

[A2.12 Current Program Status Register in AArch32 state on page A2-66](#)

[A2.13 Saved Program Status Registers in AArch32 state on page A2-67](#)

### Related reference

[A2.7 Predeclared core register names in AArch32 state on page A2-61](#)

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C2.68 MSR \(general-purpose register to PSR\) on page C2-261](#)

## A2.7 Predeclared core register names in AArch32 state

Many of the core register names have synonyms.

The following table shows the predeclared core registers:

**Table A2-2 Predeclared core registers in AArch32 state**

Register names	Meaning
r0-r15 and R0-R15	General purpose registers.
a1-a4	Argument, result or scratch registers. These are synonyms for R0 to R3.
v1-v8	Variable registers. These are synonyms for R4 to R11.
SB	Static base register. This is a synonym for R9.
IP	Intra-procedure call scratch register. This is a synonym for R12.
SP	Stack pointer. This is a synonym for R13.
LR	Link register. This is a synonym for R14.
PC	Program counter. This is a synonym for R15.

With the exception of a1-a4 and v1-v8, you can write the register names either in all upper case or all lower case.

### Related concepts

[A2.5 General-purpose registers in AArch32 state](#) on page A2-59

## A2.8 Predeclared extension register names in AArch32 state

You can write the names of Advanced SIMD and floating-point registers either in upper case or lower case.

The following table shows the predeclared extension register names:

Table A2-3 Predeclared extension registers in AArch32 state

Register names	Meaning
Q0-Q15	Advanced SIMD quadword registers
D0-D31	Advanced SIMD doubleword registers, floating-point double-precision registers
S0-S31	Floating-point single-precision registers

You can write the register names either in upper case or lower case.

## A2.9 Program Counter in AArch32 state

You can use the Program Counter explicitly, for example in some T32 data processing instructions, and implicitly, for example in branch instructions.

The *Program Counter* (PC) is accessed as PC (or R15). It is incremented by the size of the instruction executed, which is always four bytes in A32 state. Branch instructions load the destination address into the PC. You can also load the PC directly using data operation instructions. For example, to branch to the address in a general purpose register, use:

```
MOV PC,R0
```

During execution, the PC does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically PC-8 for A32, or PC-4 for T32.

————— Note ————

Arm recommends you use the BX instruction to jump to an address or to return from a function, rather than writing to the PC directly.

### Related reference

[C2.14 B on page C2-182](#)

[C2.21 BX, BXNS on page C2-192](#)

[C2.23 CBZ and CBNZ on page C2-195](#)

[C2.157 TBB and TBH on page C2-386](#)

## A2.10 The Q flag in AArch32 state

The Q flag indicates overflow or saturation. It is one of the program status flags held in the APSR.

The Q flag is set to 1 when saturation occurs in saturating arithmetic instructions, or when overflow occurs in certain multiply instructions.

The Q flag is a *sticky* flag. Although the saturating and certain multiply instructions can set the flag, they cannot clear it. You can execute a series of such instructions, and then test the flag to find out whether saturation or overflow occurred at any point in the series, without having to check the flag after each instruction.

To clear the Q flag, use an MSR instruction to read-modify-write the APSR:

```
MRS r5, APSR  
BIC r5, r5, #(1<<27)  
MSR APSR_nzcvq, r5
```

The state of the Q flag cannot be tested directly by the condition codes. To read the state of the Q flag, use an MRS instruction.

```
MRS r6, APSR  
TST r6, #(1<<27); Z is clear if Q flag was set
```

### Related reference

[C2.65 MRS \(PSR to general-purpose register\)](#) on page C2-257

[C2.68 MSR \(general-purpose register to PSR\)](#) on page C2-261

[C2.78 QADD](#) on page C2-276

[C2.128 SMULxy](#) on page C2-338

[C2.130 SMULWy](#) on page C2-340

## A2.11 Application Program Status Register

The *Application Program Status Register* (APSR) holds the program status flags that are accessible in any processor mode.

It holds copies of the N, Z, C, and V *condition flags*. The processor uses them to determine whether or not to execute conditional instructions.

The APSR also holds:

- The Q (saturation) flag.
- The APSR also holds the GE (Greater than or Equal) flags. The GE flags can be set by the parallel add and subtract instructions. They are used by the SEL instruction to perform byte-based selection from two registers.

These flags are accessible in all modes, using the MSR and MRS instructions.

### ***Related concepts***

[C1.1 Conditional instructions](#) on page C1-134

### ***Related reference***

[C1.5 Updates to the condition flags in A32/T32 code](#) on page C1-138

[C2.65 MRS \(PSR to general-purpose register\)](#) on page C2-257

[C2.68 MSR \(general-purpose register to PSR\)](#) on page C2-261

[C2.103 SEL](#) on page C2-310

## A2.12 Current Program Status Register in AArch32 state

The *Current Program Status Register* (CPSR) holds the same program status flags as the APSR, and some additional information.

It holds:

- The APSR flags.
- The processor mode.
- The interrupt disable flags.
- Either:
  - The instruction set state for the Armv8 architecture (A32 or T32).
  - The instruction set state for the Armv7 architecture (A32 or T32).
- The endianness state.
- The execution state bits for the IT block.

The execution state bits control conditional execution in the IT block.

Only the APSR flags are accessible in all modes. Arm deprecates using an `MSR` instruction to change the endianness bit (E) of the CPSR, in any mode. Each exception level can have its own endianness, but mixed endianness within an exception level is deprecated.

The `SETEND` instruction is deprecated in A32 and T32 and has no equivalent in A64.

The execution state bits for the IT block (`IT[1:0]`) and the T32 bit (T) can be accessed by `MRS` only in Debug state.

### *Related concepts*

[A2.13 Saved Program Status Registers in AArch32 state](#) on page A2-67

### *Related reference*

[C2.44 IT](#) on page C2-222

[C2.65 MRS \(PSR to general-purpose register\)](#) on page C2-257

[C2.68 MSR \(general-purpose register to PSR\)](#) on page C2-261

[C2.104 SETEND](#) on page C2-312

[C1.5 Updates to the condition flags in A32/T32 code](#) on page C1-138

## A2.13 Saved Program Status Registers in AArch32 state

The *Saved Program Status Register* (SPSR) stores the current value of the CPSR when an exception is taken so that it can be restored after handling the exception.

Each exception handling mode can access its own SPSR. User mode and System mode do not have an SPSR because they are not exception handling modes.

The execution state bits, including the endianness state and current instruction set state can be accessed from the SPSR in any exception mode, using the MSR and MRS instructions. You cannot access the SPSR using MSR or MRS in User or System mode.

### *Related concepts*

[A2.12 Current Program Status Register in AArch32 state on page A2-66](#)

## A2.14 A32 and T32 instruction set overview

A32 and T32 instructions can be grouped by functional area.

All A32 instructions are 32 bits long. Instructions are stored word-aligned, so the least significant two bits of instruction addresses are always zero in A32 state.

T32 instructions are either 16 or 32 bits long. Instructions are stored half-word aligned. Some instructions use the least significant bit of the address to determine whether the code being branched to is T32 or A32.

Before the introduction of 32-bit T32 instructions, the T32 instruction set was limited to a restricted subset of the functionality of the A32 instruction set. Almost all T32 instructions were 16-bit. Together, the 32-bit and 16-bit T32 instructions provide functionality that is almost identical to that of the A32 instruction set.

The following table describes some of the functional groupings of the available instructions.

**Table A2-4 A32 instruction groups**

<b>Instruction group</b>	<b>Description</b>
Branch and control	<p>These instructions do the following:</p> <ul style="list-style-type: none"> <li>• Branch to subroutines.</li> <li>• Branch backwards to form loops.</li> <li>• Branch forward in conditional structures.</li> <li>• Make the following instruction conditional without branching.</li> <li>• Change the processor between A32 state and T32 state.</li> </ul>
Data processing	<p>These instructions operate on the general-purpose registers. They can perform operations such as addition, subtraction, or bitwise logic on the contents of two registers and place the result in a third register. They can also operate on the value in a single register, or on a value in a register and an immediate value supplied within the instruction.</p> <p>Long multiply instructions give a 64-bit result in two registers.</p>
Register load and store	<p>These instructions load or store the value of a single register from or to memory. They can load or store a 32-bit word, a 16-bit halfword, or an 8-bit unsigned byte. Byte and halfword loads can either be sign extended or zero extended to fill the 32-bit register.</p> <p>A few instructions are also defined that can load or store 64-bit doubleword values into two 32-bit registers.</p>
Multiple register load and store	These instructions load or store any subset of the general-purpose registers from or to memory.
Status register access	These instructions move the contents of a status register to or from a general-purpose register.

## A2.15 Access to the inline barrel shifter in AArch32 state

The AArch32 arithmetic logic unit has a 32-bit barrel shifter that is capable of shift and rotate operations.

The second operand to many A32 and T32 data-processing and single register data-transfer instructions can be shifted, before the data-processing or data-transfer is executed, as part of the instruction. This supports, but is not limited to:

- Scaled addressing.
- Multiplication by an immediate value.
- Constructing immediate values.

32-bit T32 instructions give almost the same access to the barrel shifter as A32 instructions.

16-bit T32 instructions only allow access to the barrel shifter using separate instructions.



# Chapter A3

## Overview of AArch64 state

Gives an overview of the AArch64 state of Armv8.

It contains the following sections:

- [\*A3.1 Registers in AArch64 state\* on page A3-72.](#)
- [\*A3.2 Exception levels\* on page A3-73.](#)
- [\*A3.3 Link registers\* on page A3-74.](#)
- [\*A3.4 Stack Pointer register\* on page A3-75.](#)
- [\*A3.5 Predeclared core register names in AArch64 state\* on page A3-76.](#)
- [\*A3.6 Predeclared extension register names in AArch64 state\* on page A3-77.](#)
- [\*A3.7 Program Counter in AArch64 state\* on page A3-78.](#)
- [\*A3.8 Conditional execution in AArch64 state\* on page A3-79.](#)
- [\*A3.9 The Q flag in AArch64 state\* on page A3-80.](#)
- [\*A3.10 Process State\* on page A3-81.](#)
- [\*A3.11 Saved Program Status Registers in AArch64 state\* on page A3-82.](#)
- [\*A3.12 A64 instruction set overview\* on page A3-83.](#)

## A3.1 Registers in AArch64 state

Arm processors provide general-purpose and special-purpose registers. Some additional registers are available in privileged execution modes.

In AArch64 state, the following registers are available:

- Thirty-one 64-bit general-purpose registers X0-X30, the bottom halves of which are accessible as W0-W30.
- Four stack pointer registers SP\_EL0, SP\_EL1, SP\_EL2, SP\_EL3.
- Three exception link registers ELR\_EL1, ELR\_EL2, ELR\_EL3.
- Three saved program status registers SPSR\_EL1, SPSR\_EL2, SPSR\_EL3.
- One program counter.

All these registers are 64 bits wide except SPSR\_EL1, SPSR\_EL2, and SPSR\_EL3, which are 32 bits wide.

Most A64 integer instructions can operate on either 32-bit or 64-bit registers. The register width is determined by the register identifier, where W means 32-bit and X means 64-bit. The names W $n$  and X $n$ , where  $n$  is in the range 0-30, refer to the same register. When you use the 32-bit form of an instruction, the upper 32 bits of the source registers are ignored and the upper 32 bits of the destination register are set to zero.

There is no register named W31 or X31. Depending on the instruction, register 31 is either the stack pointer or the zero register. When used as the stack pointer, you refer to it as SP. When used as the zero register, you refer to it as WZR in a 32-bit context or XZR in a 64-bit context.

### ***Related concepts***

[A3.2 Exception levels on page A3-73](#)

[A3.3 Link registers on page A3-74](#)

[A3.4 Stack Pointer register on page A3-75](#)

[A3.7 Program Counter in AArch64 state on page A3-78](#)

[A3.8 Conditional execution in AArch64 state on page A3-79](#)

[A3.11 Saved Program Status Registers in AArch64 state on page A3-82](#)

## A3.2 Exception levels

The Armv8 architecture defines four exception levels, EL0 to EL3, where EL3 is the highest exception level with the most execution privilege. When taking an exception, the exception level can either increase or remain the same, and when returning from an exception, it can either decrease or remain the same.

The following is a common usage model for the exception levels:

**EL0**

Applications.

**EL1**

OS kernels and associated functions that are typically described as privileged.

**EL2**

Hypervisor.

**EL3**

Secure monitor.

When taking an exception to a higher exception level, the execution state can either remain the same, or change from AArch32 to AArch64.

When returning to a lower exception level, the execution state can either remain the same or change from AArch64 to AArch32.

The only way the execution state can change is by taking or returning from an exception. It is not possible to change between execution states in the same way as changing between A32 and T32 code in AArch32 state.

On powerup and on reset, the processor enters the highest implemented exception level. The execution state for this exception level is a property of the implementation, and might be determined by a configuration input signal.

For exception levels other than EL0, the execution state is determined by one or more control register configuration bits. These bits can be set only in a higher exception level.

For EL0, the execution state is determined as part of the exception return to EL0, under the control of the exception level that the execution is returning from.

### ***Related concepts***

[A3.3 Link registers on page A3-74](#)

[A3.11 Saved Program Status Registers in AArch64 state on page A3-82](#)

[A1.3 Changing between AArch64 and AArch32 states on page A1-50](#)

[A3.10 Process State on page A3-81](#)

## A3.3 Link registers

In AArch64 state, the Link Register (LR) stores the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored on the stack. The LR maps to register 30. Unlike in AArch32 state, the LR is distinct from the Exception Link Registers (ELRs) and is therefore unbanked.

There are three Exception Link Registers, ELR\_EL1, ELR\_EL2, and ELR\_EL3, that correspond to each of the exception levels. When an exception is taken, the Exception Link Register for the target exception level stores the return address to jump to after the handling of that exception completes. If the exception was taken from AArch32 state, the top 32 bits in the ELR are all set to zero. Subroutine calls within the exception level use the LR to store the return address from the subroutine.

For example when the exception level changes from EL0 to EL1, the return address is stored in ELR\_EL1.

When in an exception level, if you enable interrupts that use the same exception level, you must ensure you store the ELR on the stack because it will be overwritten with a new return address when the interrupt is taken.

### ***Related concepts***

[A3.7 Program Counter in AArch64 state](#) on page A3-78

### ***Related reference***

[A3.5 Predeclared core register names in AArch64 state](#) on page A3-76

## A3.4 Stack Pointer register

In AArch64 state, SP represents the 64-bit Stack Pointer. SP\_EL0 is an alias for SP. Do not use SP as a general purpose register.

You can only use SP as an operand in the following instructions:

- As the base register for loads and stores. In this case it must be quadword-aligned before adding any offset, or a stack alignment exception occurs.
- As a source or destination for arithmetic instructions, but it cannot be used as the destination in instructions that set the condition flags.
- In logical instructions, for example in order to align it.

There is a separate stack pointer for each of the three exception levels, SP\_EL1, SP\_EL2, and SP\_EL3. Within an exception level you can either use the dedicated stack pointer for that exception level or you can use SP\_EL0, the stack pointer associated with EL0. You can use the SPSel register to select which stack pointer to use in the exception level.

The choice of stack pointer is indicated by the letter t or h appended to the exception level name, for example EL0t or EL3h. The t suffix indicates that the exception level uses SP\_EL0 and the h suffix indicates it uses SP\_ELx, where x is the current exception level number. EL0 always uses SP\_EL0 so cannot have an h suffix.

### ***Related concepts***

[A3.2 Exception levels on page A3-73](#)

[A3.10 Process State on page A3-81](#)

### ***Related reference***

[A3.1 Registers in AArch64 state on page A3-72](#)

## A3.5 Predeclared core register names in AArch64 state

In AArch64 state, the predeclared core registers are different from those in AArch32 state.

The following table shows the predeclared core registers in AArch64 state:

**Table A3-1 Predeclared core registers in AArch64 state**

Register names	Meaning
W0-W30	32-bit general purpose registers.
X0-X30	64-bit general purpose registers.
WZR	32-bit RAZ/WI register. This is the name for register 31 when it is used as the zero register in a 32-bit context.
XZR	64-bit RAZ/WI register. This is the name for register 31 when it is used as the zero register in a 64-bit context.
WSP	32-bit stack pointer. This is the name for register 31 when it is used as the stack pointer in a 32-bit context.
SP	64-bit stack pointer. This is the name for register 31 when it is used as the stack pointer in a 64-bit context.
LR	Link register. This is a synonym for X30.

You can write the register names either in all upper case or all lower case.

————— **Note** —————

In AArch64 state, the PC is not a general purpose register and you cannot access it by name.

**Related concepts**

[A3.3 Link registers on page A3-74](#)

[A3.4 Stack Pointer register on page A3-75](#)

[A3.7 Program Counter in AArch64 state on page A3-78](#)

**Related reference**

[A3.1 Registers in AArch64 state on page A3-72](#)

## A3.6 Predeclared extension register names in AArch64 state

You can write the names of Advanced SIMD and floating-point registers either in upper case or lower case.

The following table shows the predeclared extension register names in AArch64 state:

**Table A3-2 Predeclared extension registers in AArch64 state**

Register names	Meaning
V0-V31	Advanced SIMD 128-bit vector registers.
Q0-Q31	Advanced SIMD registers holding a 128-bit scalar.
D0-D31	Advanced SIMD registers holding a 64-bit scalar, floating-point double-precision registers.
S0-S31	Advanced SIMD registers holding a 32-bit scalar, floating-point single-precision registers.
H0-H31	Advanced SIMD registers holding a 16-bit scalar, floating-point half-precision registers.
B0-B31	Advanced SIMD registers holding an 8-bit scalar.

**Related reference**

[A3.1 Registers in AArch64 state on page A3-72](#)

## A3.7 Program Counter in AArch64 state

In AArch64 state, the *Program Counter* (PC) contains the address of the currently executing instruction. It is incremented by the size of the instruction executed, which is always four bytes.

In AArch64 state, the PC is not a general purpose register and you cannot access it explicitly. The following types of instructions read it implicitly:

- Instructions that compute a PC-relative address.
- PC-relative literal loads.
- Direct branches to a PC-relative label.
- Branch and link instructions, which store it in the procedure link register.

The only types of instructions that can write to the PC are:

- Conditional and unconditional branches.
- Exception generation and exception returns.

Branch instructions load the destination address into the PC.

### ***Related reference***

[D2.27 B.cond on page D2-697](#)

[D2.28 B on page D2-698](#)

[D2.35 BL on page D2-705](#)

## A3.8 Conditional execution in AArch64 state

In AArch64 state, the NZCV register holds copies of the N, Z, C, and V *condition flags*. The processor uses them to determine whether or not to execute conditional instructions. The NZCV register contains the flags in bits[31:28].

The condition flags are accessible in all exception levels, using the MSR and MRS instructions.

A64 makes less use of conditionality than A32. For example, in A64:

- Only a few instructions can set or test the condition flags.
- There is no equivalent of the T32 IT instruction.
- The only conditionally executed instruction, which behaves as a NOP if the condition is false, is the conditional branch, B.cond.

### **Related reference**

[D1.3 Updates to the condition flags in A64 code](#) on page D1-650

[D2.109 MRS](#) on page D2-784

[D2.111 MSR \(register\)](#) on page D2-786

## A3.9 The Q flag in AArch64 state

In AArch64 state, you cannot read or write to the Q flag because in A64 there are no saturating arithmetic instructions that operate on the general purpose registers.

The Advanced SIMD saturating arithmetic instructions set the QC bit in the floating-point status register (FPSR) to indicate that saturation has occurred. You can identify such instructions by the Q mnemonic modifier, for example SQADD.

### *Related reference*

*Chapter D5 A64 SIMD Scalar Instructions* on page D5-1107

*Chapter D6 A64 SIMD Vector Instructions* on page D6-1237

## A3.10 Process State

In AArch64 state, there is no *Current Program Status Register* (CPSR). You can access the different components of the traditional CPSR independently as *Process State* fields.

The Process State fields are:

- N, Z, C, and V condition flags (NZCV).
- Current register width (nRW).
- Stack pointer selection bit (SPSel).
- Interrupt disable flags (DAIF).
- Current exception level (EL).
- Single step process state bit (SS).
- Illegal exception return state bit (IL).

You can use MSR to write to:

- The N, Z, C, and V flags in the NZCV register.
- The interrupt disable flags in the DAIF register.
- The SP selection bit in the SPSel register, in EL1 or higher.

You can use MRS to read:

- The N, Z, C, and V flags in the NZCV register.
- The interrupt disable flags in the DAIF register.
- The exception level bits in the CurrentEL register, in EL1 or higher.
- The SP selection bit in the SPSel register, in EL1 or higher.

When an exception occurs, all Process State fields associated with the current exception level are stored in a single register associated with the target exception level, the SPSR. You can access the SS, IL, and nRW bits only from the SPSR.

### *Related concepts*

[A3.11 Saved Program Status Registers in AArch64 state](#) on page A3-82

### *Related reference*

[D1.3 Updates to the condition flags in A64 code](#) on page D1-650

[D2.109 MRS](#) on page D2-784

[D2.111 MSR \(register\)](#) on page D2-786

## A3.11 Saved Program Status Registers in AArch64 state

The *Saved Program Status Registers* (SPSRs) are 32-bit registers that store the process state of the current exception level when an exception is taken to an exception level that uses AArch64 state. This allows the process state to be restored after the exception has been handled.

In AArch64 state, each target exception level has its own SPSR:

- SPSR\_EL1.
- SPSR\_EL2.
- SPSR\_EL3.

When taking an exception, the process state of the current exception level is stored in the SPSR of the target exception level. On returning from an exception, the exception handler uses the SPSR of the exception level that is being returned from to restore the process state of the exception level that is being returned to.

————— Note ————

On returning from an exception, the preferred return address is restored from the ELR associated with the exception level that is being returned from.

The SPSRs store the following information:

- N, Z, C, and V flags.
- D, A, I, and F interrupt disable bits.
- The register width.
- The execution mode.
- The IL and SS bits.

**Related concepts**

[A3.4 Stack Pointer register on page A3-75](#)

[A3.10 Process State on page A3-81](#)

## A3.12 A64 instruction set overview

A64 instructions can be grouped by functional area.

The following table describes some of the functional groupings of the instructions in A64.

**Table A3-3 A64 instruction groups**

Instruction group	Description
Branch and control	<p>These instructions do the following:</p> <ul style="list-style-type: none"><li>• Branch to and return from subroutines.</li><li>• Branch backwards to form loops.</li><li>• Branch forward in conditional structures.</li><li>• Generate and return from exceptions.</li></ul>
Data processing	<p>These instructions operate on the general-purpose registers. They can perform operations such as addition, subtraction, or bitwise logic on the contents of two registers and place the result in a third register. They can also operate on the value in a single register, or on a value in a register and an immediate value supplied within the instruction.</p> <p>The addition and subtraction instructions can optionally left shift the immediate operand, or can sign or zero-extend and shift the final source operand register.</p> <p>A64 includes signed and unsigned 32-bit and 64-bit multiply and divide instructions.</p>
Register load and store	<p>These instructions load or store the value of a single register or pair of registers from or to memory. You can load or store a single 64-bit doubleword, 32-bit word, 16-bit halfword, or 8-bit byte, or a pair of words or doublewords. Byte and halfword loads can either be sign-extended or zero-extended to fill the 32-bit register. You can also load and sign-extend a signed byte, halfword or word into a 64-bit register, or load a pair of signed words into two 64-bit registers.</p>
System register access	<p>These instructions move the contents of a system register to or from a general-purpose register.</p>

### **Related reference**

*Chapter D2 A64 General Instructions* on page D2-657

*Chapter D3 A64 Data Transfer Instructions* on page D3-873



## **Part B**

# **Advanced SIMD and Floating-point Programming**



# Chapter B1

## Advanced SIMD Programming

Describes Advanced SIMD assembly language programming.

It contains the following sections:

- [\*B1.1 Architecture support for Advanced SIMD\* on page B1-88.](#)
- [\*B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state\* on page B1-89.](#)
- [\*B1.3 Extension register bank mapping for Advanced SIMD in AArch64 state\* on page B1-91.](#)
- [\*B1.4 Views of the Advanced SIMD register bank in AArch32 state\* on page B1-93.](#)
- [\*B1.5 Views of the Advanced SIMD register bank in AArch64 state\* on page B1-94.](#)
- [\*B1.6 Differences between A32/T32 and A64 Advanced SIMD instruction syntax\* on page B1-95.](#)
- [\*B1.7 Load values to Advanced SIMD registers\* on page B1-97.](#)
- [\*B1.8 Conditional execution of A32/T32 Advanced SIMD instructions\* on page B1-98.](#)
- [\*B1.9 Floating-point exceptions for Advanced SIMD in A32/T32 instructions\* on page B1-99.](#)
- [\*B1.10 Advanced SIMD data types in A32/T32 instructions\* on page B1-100.](#)
- [\*B1.11 Polynomial arithmetic over {0,1}\* on page B1-101.](#)
- [\*B1.12 Advanced SIMD vectors\* on page B1-102.](#)
- [\*B1.13 Normal, long, wide, and narrow Advanced SIMD instructions\* on page B1-103.](#)
- [\*B1.14 Saturating Advanced SIMD instructions\* on page B1-104.](#)
- [\*B1.15 Advanced SIMD scalars\* on page B1-105.](#)
- [\*B1.16 Extended notation extension for Advanced SIMD in A32/T32 code\* on page B1-106.](#)
- [\*B1.17 Advanced SIMD system registers in AArch32 state\* on page B1-107.](#)
- [\*B1.18 Flush-to-zero mode in Advanced SIMD\* on page B1-108.](#)
- [\*B1.19 When to use flush-to-zero mode in Advanced SIMD\* on page B1-109.](#)
- [\*B1.20 The effects of using flush-to-zero mode in Advanced SIMD\* on page B1-110.](#)
- [\*B1.21 Advanced SIMD operations not affected by flush-to-zero mode\* on page B1-111.](#)

## B1.1 Architecture support for Advanced SIMD

Advanced SIMD is an optional extension to the Armv8 and Armv7 architectures.

All Advanced SIMD instructions are available on systems that support Advanced SIMD. In A32, some of these instructions are also available on systems that implement the floating-point extension without Advanced SIMD. These are called shared instructions.

In AArch32 state, the Advanced SIMD register bank consists of thirty-two 64-bit registers, and smaller registers are packed into larger ones, as in Armv7.

In AArch64 state, the Advanced SIMD register bank includes thirty-two 128-bit registers and has a new register packing model.

————— **Note** —————

Advanced SIMD and floating-point instructions share the same extension register bank.

Advanced SIMD instructions in A64 are closely based on VFPv4 and A32, but with new instruction mnemonics and some functional enhancements.

***Related information***

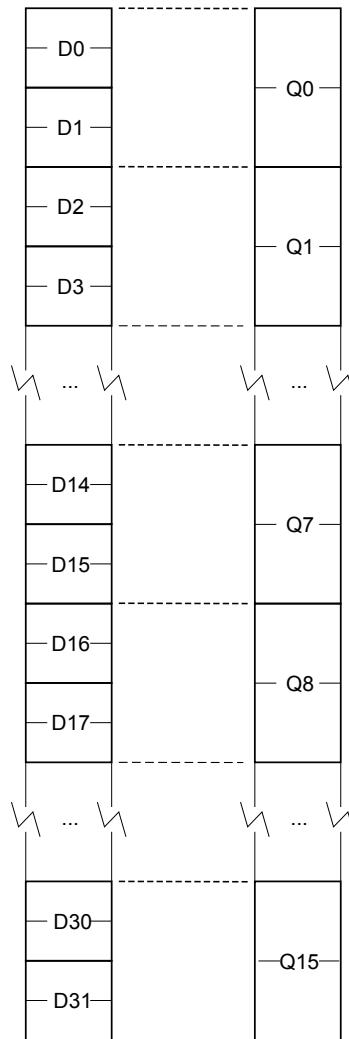
*Floating-point support*

## B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state

The Advanced SIMD extension register bank is a collection of registers that can be accessed as either 64-bit or 128-bit registers.

Advanced SIMD and floating-point instructions use the same extension register bank, and is distinct from the Arm core register bank.

The following figure shows the views of the extension register bank, and the overlap between the different size registers. For example, the 128-bit register Q0 is an alias for two consecutive 64-bit registers D0 and D1. The 128-bit register Q8 is an alias for 2 consecutive 64-bit registers D16 and D17.



**Figure B1-1 Extension register bank for Advanced SIMD in AArch32 state**

---

**Note**

If your processor supports both Advanced SIMD and floating-point, all the Advanced SIMD registers overlap with the floating-point registers.

The aliased views enable half-precision, single-precision, and double-precision values, and Advanced SIMD vectors to coexist in different non-overlapped registers at the same time.

You can also use the same overlapped registers to store half-precision, single-precision, and double-precision values, and Advanced SIMD vectors at different times.

Do not attempt to use overlapped 64-bit and 128-bit registers at the same time because it creates meaningless results.

The mapping between the registers is as follows:

- $D_{2n}$  maps to the least significant half of  $Q_n$
- $D_{2n+1}$  maps to the most significant half of  $Q_n$ .

For example, you can access the least significant half of the elements of a vector in Q6 by referring to D12, and the most significant half of the elements by referring to D13.

#### ***Related concepts***

[B1.3 Extension register bank mapping for Advanced SIMD in AArch64 state on page B1-91](#)

[B2.4 Views of the floating-point extension register bank in AArch32 state on page B2-118](#)

[B1.4 Views of the Advanced SIMD register bank in AArch32 state on page B1-93](#)

## B1.3 Extension register bank mapping for Advanced SIMD in AArch64 state

The extension register bank is a collection of registers that can be accessed as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit.

Advanced SIMD and floating-point instructions use the same extension register bank, and is distinct from the Arm core register bank.

The following figure shows the views of the extension register bank, and the overlap between the different size registers.

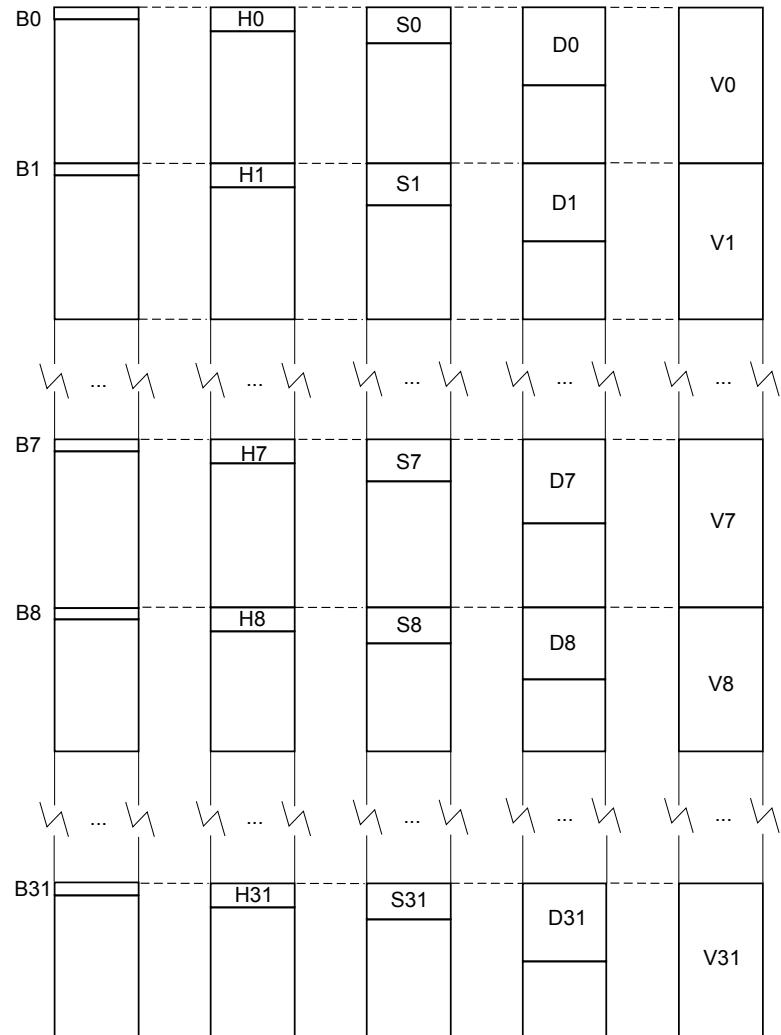


Figure B1-2 Extension register bank for Advanced SIMD in AArch64 state

The mapping between the registers is as follows:

- $D_{<n>}$  maps to the least significant half of  $V_{<n>}$
- $S_{<n>}$  maps to the least significant half of  $D_{<n>}$
- $H_{<n>}$  maps to the least significant half of  $S_{<n>}$
- $B_{<n>}$  maps to the least significant half of  $H_{<n>}$ .

For example, you can access the least significant half of the elements of a vector in  $V7$  by referring to  $D7$ .

Registers Q0-Q31 map directly to registers V0-V31.

### Related concepts

[B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page B1-89](#)

*B2.4 Views of the floating-point extension register bank in AArch32 state* on page B2-118

*B1.4 Views of the Advanced SIMD register bank in AArch32 state* on page B1-93

## B1.4 Views of the Advanced SIMD register bank in AArch32 state

Advanced SIMD can have different views of the extension register bank in AArch32 state.

It can view the extension register bank as:

- Sixteen 128-bit registers, Q0-Q15.
- Thirty-two 64-bit registers, D0-D31.
- A combination of registers from these views.

Advanced SIMD views each register as containing a *vector* of 1, 2, 4, 8, or 16 elements, all of the same size and type. Individual elements can also be accessed as *scalars*.

In Advanced SIMD, the 64-bit registers are called doubleword registers and the 128-bit registers are called quadword registers.

### **Related concepts**

[B1.5 Views of the Advanced SIMD register bank in AArch64 state on page B1-94](#)

[B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page B1-89](#)

[B2.4 Views of the floating-point extension register bank in AArch32 state on page B2-118](#)

## B1.5 Views of the Advanced SIMD register bank in AArch64 state

Advanced SIMD can have different views of the extension register bank in AArch64 state.

It can view the extension register bank as:

- Thirty-two 128-bit registers V0-V31.
- Thirty-two 64-bit registers D0-D31.
- Thirty-two 32-bit registers S0-S31.
- Thirty-two 16-bit registers H0-H31.
- Thirty-two 8-bit registers B0-B31.
- A combination of registers from these views.

### *Related concepts*

[B1.4 Views of the Advanced SIMD register bank in AArch32 state on page B1-93](#)

[B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page B1-89](#)

[B2.4 Views of the floating-point extension register bank in AArch32 state on page B2-118](#)

## B1.6 Differences between A32/T32 and A64 Advanced SIMD instruction syntax

The syntax and mnemonics of A64 Advanced SIMD instructions are based on those in A32/T32 but with some differences.

The following table describes the main differences.

**Table B1-1 Differences in syntax and mnemonics between A32/T32 and A64 Advanced SIMD instructions**

A32/T32	A64
All Advanced SIMD instruction mnemonics begin with V, for example VMAX.	The first letter of the instruction mnemonic indicates the data type of the instruction. For example, SMAX, UMAX, and FMAX mean signed, unsigned, and floating-point respectively. No suffix means the type is irrelevant and P means polynomial.
A mnemonic qualifier specifies the type and width of elements in a vector. For example, in the following instruction, U32 means 32-bit unsigned integers:  VMAX.U32 Q0, Q1, Q2	A register qualifier specifies the data width and the number of elements in the register. For example, in the following instruction .4S means 4 32-bit elements:  UMAX V0.4S, V1.4S, V2.4S
The 128-bit vector registers are named Q0-Q15 and the 64-bit vector registers are named D0-D31.	All vector registers are named Vn , where n is a register number between 0 and 31. You only use one of the qualified register names Qn, Dn, Sn, Hn or Bn when referring to a scalar register, to indicate the number of significant bits.
You load a single element into one or more vector registers by appending an index to each register individually, for example:  VLD4.8 {D0[3], D1[3], D2[3], D3[3]}, [R0]	You load a single element into one or more vector registers by appending the index to the register list, for example:  LD4 {V0.B, V1.B, V2.B, V3.B}[3], [X0]
You can append a condition code to most Advanced SIMD instruction mnemonics to make them conditional.	A64 has no conditionally executed floating-point or Advanced SIMD instructions.
L, W and N suffixes indicate long, wide and narrow variants of Advanced SIMD data processing instructions. A32/T32 Advanced SIMD does not include vector narrowing or widening second part instructions.	L, W and N suffixes indicate long, wide and narrow variants of Advanced SIMD data processing instructions. You can additionally append a 2 to implement the second part of a narrowing or widening operation, for example:  UADDL2 V0.4S, V1.8H, V2.8H ; take input from 4 high-numbered lanes of V1 and V2
A32/T32 Advanced SIMD does not include vector reduction instructions.	The V Advanced SIMD mnemonic suffix identifies vector reduction instructions, in which the operand is a vector and the result a scalar, for example:  ADDV S0, V1.4S
The P mnemonic qualifier which indicates pairwise instructions is a prefix, for example, VPADD.	The P mnemonic qualifier is a suffix, for example ADDP.

### Related concepts

[B1.10 Advanced SIMD data types in A32/T32 instructions](#) on page B1-100

[B1.8 Conditional execution of A32/T32 Advanced SIMD instructions](#) on page B1-98

[B1.15 Advanced SIMD scalars](#) on page B1-105

[B1.13 Normal, long, wide, and narrow Advanced SIMD instructions](#) on page B1-103

**Related reference**

[C4.36 VSEL on page C4-637](#)

[D4.9 FCSEL on page D4-1042](#)

## B1.7 Load values to Advanced SIMD registers

To load a register with a floating-point immediate value, use VMOV in A32 or FMOV in A64. Both instructions exist in scalar and vector forms.

The A32 Advanced SIMD instructions VMOV and VMVN can also load integer immediates. The A64 Advanced SIMD instructions to load integer immediates are MOVI and MVNI.

### ***Related reference***

[C3.57 VLDR pseudo-instruction](#) on page C3-506

[C4.22 VMOV \(floating-point\)](#) on page C4-623

[C3.68 VMOV \(immediate\)](#) on page C3-517

## B1.8 Conditional execution of A32/T32 Advanced SIMD instructions

Most Advanced SIMD instructions always execute unconditionally.

You cannot use any of the following Advanced SIMD instructions in an IT block:

- VCVT{A, N, P, M}.
- VMAXNM.
- VMINNM.
- VRINT{N, X, A, Z, M, P}.
- All instructions in the Crypto extension.

In addition, specifying any other Advanced SIMD instruction in an IT block is deprecated.

Arm deprecates conditionally executing any Advanced SIMD instruction unless it is a shared Advanced SIMD and floating-point instruction.

### ***Related concepts***

[C1.2 Conditional execution in A32 code](#) on page C1-135

[C1.3 Conditional execution in T32 code](#) on page C1-136

### ***Related reference***

[C1.11 Comparison of condition code meanings in integer and floating-point code](#) on page C1-144

[C1.9 Condition code suffixes](#) on page C1-142

## B1.9 Floating-point exceptions for Advanced SIMD in A32/T32 instructions

The Advanced SIMD extension records floating-point exceptions in the FPSCR cumulative flags.

It records the following exceptions:

### Invalid operation

The exception is caused if the result of an operation has no mathematical value or cannot be represented.

### Division by zero

The exception is caused if a divide operation has a zero divisor and a dividend that is not zero, an infinity or a NaN.

### Overflow

The exception is caused if the absolute value of the result of an operation, produced after rounding, is greater than the maximum positive normalized number for the destination precision.

### Underflow

The exception is caused if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, and the rounded result is inexact.

### Inexact

The exception is caused if the result of an operation is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

### Input denormal

The exception is caused if a denormalized input operand is replaced in the computation by a zero.

The descriptions of the Advanced SIMD instructions that can cause floating-point exceptions include a subsection listing the exceptions. If there is no such subsection, that instruction cannot cause any floating-point exception.

### Related concepts

[B1.18 Flush-to-zero mode in Advanced SIMD](#) on page B1-108

### Related reference

[Chapter B1 Advanced SIMD Programming](#) on page B1-87

### Related information

[Arm Architecture Reference Manual](#)

## B1.10 Advanced SIMD data types in A32/T32 instructions

Most Advanced SIMD instructions use a data type specifier to define the size and type of data that the instruction operates on.

Data type specifiers in Advanced SIMD instructions consist of a letter indicating the type of data, usually followed by a number indicating the width. They are separated from the instruction mnemonic by a point. The following table shows the data types available in Advanced SIMD instructions:

**Table B1-2 Advanced SIMD data types**

	8-bit	16-bit	32-bit	64-bit
Unsigned integer	U8	U16	U32	U64
Signed integer	S8	S16	S32	S64
Integer of unspecified type	I8	I16	I32	I64
Floating-point number	not available	F16	F32 (or F)	not available
Polynomial over {0,1}	P8	P16	not available	not available

The datatype of the second (or only) operand is specified in the instruction.

— Note —

Most instructions have a restricted range of permitted data types. See the instruction descriptions for details. However, the data type description is flexible:

- If the description specifies I, you can also use the S or U data types.
- If only the data size is specified, you can specify a type (I, S, U, P or F).
- If no data type is specified, you can specify a data type.

**Related concepts**

[B1.10 Advanced SIMD data types in A32/T32 instructions](#) on page B1-100

[B1.11 Polynomial arithmetic over {0,1}](#) on page B1-101

## B1.11 Polynomial arithmetic over {0,1}

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic.

The following rules apply:

- $0 + 0 = 1 + 1 = 0$ .
- $0 + 1 = 1 + 0 = 1$ .
- $0 * 0 = 0 * 1 = 1 * 0 = 0$ .
- $1 * 1 = 1$ .

That is, adding two polynomials over {0,1} is the same as a bitwise exclusive OR, and multiplying two polynomials over {0,1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

### *Related concepts*

[B1.10 Advanced SIMD data types in A32/T32 instructions](#) on page B1-100

## B1.12 Advanced SIMD vectors

An Advanced SIMD operand can be a vector or a scalar. An Advanced SIMD vector can be a 64-bit doubleword vector or a 128-bit quadword vector.

In A32/T32 Advanced SIMD instructions, the size of the elements in an Advanced SIMD vector is specified by a datatype suffix appended to the mnemonic. In A64 Advanced SIMD instructions, the size and number of the elements in an Advanced SIMD vector are specified by a suffix appended to the register.

Doubleword vectors can contain:

- Eight 8-bit elements.
- Four 16-bit elements.
- Two 32-bit elements.
- One 64-bit element.

Quadword vectors can contain:

- Sixteen 8-bit elements.
- Eight 16-bit elements.
- Four 32-bit elements.
- Two 64-bit elements.

### *Related concepts*

[B1.15 Advanced SIMD scalars](#) on page B1-105

[B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state](#) on page B1-89

[B1.16 Extended notation extension for Advanced SIMD in A32/T32 code](#) on page B1-106

[B1.10 Advanced SIMD data types in A32/T32 instructions](#) on page B1-100

[B1.13 Normal, long, wide, and narrow Advanced SIMD instructions](#) on page B1-103

## B1.13 Normal, long, wide, and narrow Advanced SIMD instructions

Many A32/T32 and A64 Advanced SIMD data processing instructions are available in Normal, Long, Wide, Narrow, and saturating variants.

### Normal operation

The operands can be any of the vector types. The result vector is the same width, and usually the same type, as the operand vectors, for example:

VADD.I16 D0, D1, D2

You can specify that the operands and result of a normal A32/T32 Advanced SIMD instruction must all be quadwords by appending a Q to the instruction mnemonic. If you do this, `armasm` produces an error if the operands or result are not quadwords.

### Long operation

The operands are doubleword vectors and the result is a quadword vector. The elements of the result are usually twice the width of the elements of the operands, and the same type.

Long operation is specified using an L appended to the instruction mnemonic, for example:

VADDL.S16 Q0, D2, D3

### Wide operation

One operand vector is doubleword and the other is quadword. The result vector is quadword. The elements of the result and the first operand are twice the width of the elements of the second operand.

Wide operation is specified using a W appended to the instruction mnemonic, for example:

VADDW.S16 Q0, Q1, D4

### Narrow operation

The operands are quadword vectors and the result is a doubleword vector. The elements of the result are half the width of the elements of the operands.

Narrow operation is specified using an N appended to the instruction mnemonic, for example:

VADDHN.I16 D0, Q1, Q2

### *Related concepts*

[B1.12 Advanced SIMD vectors on page B1-102](#)

## B1.14 Saturating Advanced SIMD instructions

Saturating instructions saturate the result to the value of the upper limit or lower limit if the result overflows or underflows.

The saturation limits depend on the datatype of the instruction. The following table shows the ranges that Advanced SIMD saturating instructions saturate to, where  $x$  is the result of the operation.

**Table B1-3 Advanced SIMD saturation ranges**

Data type	Saturation range of $x$
Signed byte (S8)	$-2^7 \leq x < 2^7$
Signed halfword (S16)	$-2^{15} \leq x < 2^{15}$
Signed word (S32)	$-2^{31} \leq x < 2^{31}$
Signed doubleword (S64)	$-2^{63} \leq x < 2^{63}$
Unsigned byte (U8)	$0 \leq x < 2^8$
Unsigned halfword (U16)	$0 \leq x < 2^{16}$
Unsigned word (U32)	$0 \leq x < 2^{32}$
Unsigned doubleword (U64)	$0 \leq x < 2^{64}$

Saturating Advanced SIMD arithmetic instructions set the QC bit in the floating-point status register (FPSCR in AArch32 or FPSR in AArch64) to indicate that saturation has occurred.

Saturating instructions are specified using a Q prefix. In A32/T32 Advanced SIMD instructions, this is inserted between the V and the instruction mnemonic, or between the S or U and the mnemonic in A64 Advanced SIMD instructions.

### Related reference

[C2.7 Saturating instructions](#) on page C2-168

## B1.15 Advanced SIMD scalars

Some Advanced SIMD instructions act on scalars in combination with vectors. Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit.

In A32/T32 Advanced SIMD instructions, the instruction syntax refers to a single element in a vector register using an index,  $x$ , into the vector, so that  $Dm[x]$  is the  $x$ th element in vector  $Dm$ . In A64 Advanced SIMD instructions, you append the index to the element size specifier, so that  $Vm.D[x]$  is the  $x$ th doubleword element in vector  $Vm$ .

In A64 Advanced SIMD scalar instructions, you refer to registers using a name that indicates the number of significant bits. The names are  $Bn$ ,  $Hn$ ,  $Sn$ , or  $Dn$ , where  $n$  is the register number (0-31). The unused high bits are ignored on a read and set to zero on a write.

Other than A32/T32 Advanced SIMD multiply instructions, instructions that access scalars can access any element in the register bank.

A32/T32 Advanced SIMD multiply instructions only allow 16-bit or 32-bit scalars, and can only access the first 32 scalars in the register bank. That is, in multiply instructions:

- 16-bit scalars are restricted to registers D0-D7, with  $x$  in the range 0-3.
- 32-bit scalars are restricted to registers D0-D15, with  $x$  either 0 or 1.

### ***Related concepts***

[B1.12 Advanced SIMD vectors on page B1-102](#)

[B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page B1-89](#)

## B1.16 Extended notation extension for Advanced SIMD in A32/T32 code

`armasm` implements an extension to the architectural Advanced SIMD assembly syntax, called *extended notation*. This extension allows you to include datatype information or scalar indexes in register names.

————— Note —————

Extended notation is not supported for A64 code.

If you use extended notation, you do not have to include the data type or scalar index information in every instruction.

Register names can be any of the following:

**Untyped**

The register name specifies the register, but not what datatype it contains, nor any index to a particular scalar within the register.

**Untyped with scalar index**

The register name specifies the register, but not what datatype it contains, It specifies an index to a particular scalar within the register.

**Typed**

The register name specifies the register, and what datatype it contains, but not any index to a particular scalar within the register.

**Typed with scalar index**

The register name specifies the register, what datatype it contains, and an index to a particular scalar within the register.

Use the `DN` and `QN` directives to define names for typed and scalar registers.

**Related concepts**

[B1.12 Advanced SIMD vectors on page B1-102](#)

[B1.10 Advanced SIMD data types in A32/T32 instructions on page B1-100](#)

[B1.15 Advanced SIMD scalars on page B1-105](#)

## B1.17 Advanced SIMD system registers in AArch32 state

Advanced SIMD system registers are accessible in all implementations of Advanced SIMD.

For exception levels using AArch32, the following Advanced SIMD system registers are accessible in all Advanced SIMD implementations:

- FPSCR, the floating-point status and control register.
- FPEXC, the floating-point exception register.
- FPSID, the floating-point system ID register.

A particular Advanced SIMD implementation can have additional registers. For more information, see the Technical Reference Manual for your processor.

————— Note ————

Advanced SIMD technology shares the same set of system registers as floating-point.

**Related information**

*Arm Architecture Reference Manual*

## B1.18 Flush-to-zero mode in Advanced SIMD

Flush-to-zero mode replaces denormalized numbers with zero. This does not comply with IEEE 754 arithmetic, but in some circumstances can improve performance considerably.

Flush-to-zero mode in Advanced SIMD always preserves the sign bit.

Advanced SIMD always uses flush-to-zero mode.

### ***Related concepts***

[B1.20 The effects of using flush-to-zero mode in Advanced SIMD](#) on page B1-110

### ***Related reference***

[B1.19 When to use flush-to-zero mode in Advanced SIMD](#) on page B1-109

[B1.21 Advanced SIMD operations not affected by flush-to-zero mode](#) on page B1-111

## B1.19 When to use flush-to-zero mode in Advanced SIMD

You can change between flush-to-zero mode and normal mode, depending on the requirements of different parts of your code.

You must select flush-to-zero mode if all the following are true:

- IEEE 754 compliance is not a requirement for your system.
- The algorithms you are using sometimes generate denormalized numbers.
- Your system uses support code to handle denormalized numbers.
- The algorithms you are using do not depend for their accuracy on the preservation of denormalized numbers.
- The algorithms you are using do not generate frequent exceptions as a result of replacing denormalized numbers with 0.

You select flush-to-zero mode in one of the following ways:

- In A32 code, by setting the FZ bit in the FPSCR to 1. You do this using the VMRS and VMSR instructions.
- In A64 code, by setting the FZ bit in the FPCR to 1. You do this using the MRS and MSR instructions.

You can change between flush-to-zero and normal mode at any time, if different parts of your code have different requirements. Numbers already in registers are not affected by changing mode.

### ***Related concepts***

[B1.18 Flush-to-zero mode in Advanced SIMD on page B1-108](#)

[B1.20 The effects of using flush-to-zero mode in Advanced SIMD on page B1-110](#)

## B1.20 The effects of using flush-to-zero mode in Advanced SIMD

In flush-to-zero mode, denormalized inputs are treated as zero. Results that are too small to be represented in a normalized number are replaced with zero.

With certain exceptions, flush-to-zero mode has the following effects on floating-point operations:

- A denormalized number is treated as 0 when used as an input to a floating-point operation. The source register is not altered.
- If the result of a single-precision floating-point operation, before rounding, is in the range  $-2^{-126}$  to  $+2^{-126}$ , it is replaced by 0.
- If the result of a double-precision floating-point operation, before rounding, is in the range  $-2^{-1022}$  to  $+2^{-1022}$ , it is replaced by 0.

In flush-to-zero mode, an Input Denormal exception occurs whenever a denormalized number is used as an operand. An Underflow exception occurs when a result is flushed-to-zero.

### *Related concepts*

[B1.18 Flush-to-zero mode in Advanced SIMD](#) on page B1-108

### *Related reference*

[B1.21 Advanced SIMD operations not affected by flush-to-zero mode](#) on page B1-111

## B1.21 Advanced SIMD operations not affected by flush-to-zero mode

Some Advanced SIMD instructions can be carried out on denormalized numbers even in flush-to-zero mode, without flushing the results to zero.

These instructions are as follows:

- Copy, absolute value, and negate (VMOV, VMVN, V{Q}ABS, and V{Q}NEG).
- Duplicate (VDUP).
- Swap (VSWP).
- Load and store (VLDR and VSTR).
- Load multiple and store multiple (VLDM and VSTM).
- Transfer between extension registers and AArch32 general-purpose registers (VMOV).

### ***Related concepts***

[B1.18 Flush-to-zero mode in Advanced SIMD](#) on page B1-108

### ***Related reference***

[C3.9 VABS](#) on page C3-455

[C4.2 VABS \(floating-point\)](#) on page C4-603

[C3.41 VDUP](#) on page C3-487

[C3.54 VLDM](#) on page C3-503

[C3.55 VLDR](#) on page C3-504

[C3.69 VMOV \(register\)](#) on page C3-518

[C3.70 VMOV \(between two general-purpose registers and a 64-bit extension register\)](#) on page C3-519

[C3.71 VMOV \(between a general-purpose register and an Advanced SIMD scalar\)](#) on page C3-520

[C3.139 VSWP](#) on page C3-590



# Chapter B2

## Floating-point Programming

Describes floating-point assembly language programming.

It contains the following sections:

- [\*B2.1 Architecture support for floating-point\*](#) on page B2-114.
- [\*B2.2 Extension register bank mapping for floating-point in AArch32 state\*](#) on page B2-115.
- [\*B2.3 Extension register bank mapping in AArch64 state\*](#) on page B2-117.
- [\*B2.4 Views of the floating-point extension register bank in AArch32 state\*](#) on page B2-118.
- [\*B2.5 Views of the floating-point extension register bank in AArch64 state\*](#) on page B2-119.
- [\*B2.6 Differences between A32/T32 and A64 floating-point instruction syntax\*](#) on page B2-120.
- [\*B2.7 Load values to floating-point registers\*](#) on page B2-121.
- [\*B2.8 Conditional execution of A32/T32 floating-point instructions\*](#) on page B2-122.
- [\*B2.9 Floating-point exceptions for floating-point in A32/T32 instructions\*](#) on page B2-123.
- [\*B2.10 Floating-point data types in A32/T32 instructions\*](#) on page B2-124.
- [\*B2.11 Extended notation extension for floating-point in A32/T32 code\*](#) on page B2-125.
- [\*B2.12 Floating-point system registers in AArch32 state\*](#) on page B2-126.
- [\*B2.13 Flush-to-zero mode in floating-point\*](#) on page B2-127.
- [\*B2.14 When to use flush-to-zero mode in floating-point\*](#) on page B2-128.
- [\*B2.15 The effects of using flush-to-zero mode in floating-point\*](#) on page B2-129.
- [\*B2.16 Floating-point operations not affected by flush-to-zero mode\*](#) on page B2-130.

## B2.1 Architecture support for floating-point

Floating-point is an optional extension to the Arm architecture. There are versions that provide additional instructions.

The floating-point instruction set supported in A32 is based on VFPv4, but with the addition of some new instructions, including the following:

- Floating-point round to integral.
- Conversion from floating-point to integer with a directed rounding mode.
- Direct conversion between half-precision and double-precision floating-point.
- Floating-point conditional select.

In AArch32 state, the register bank consists of thirty-two 64-bit registers, and smaller registers are packed into larger ones, as in Armv7 and earlier.

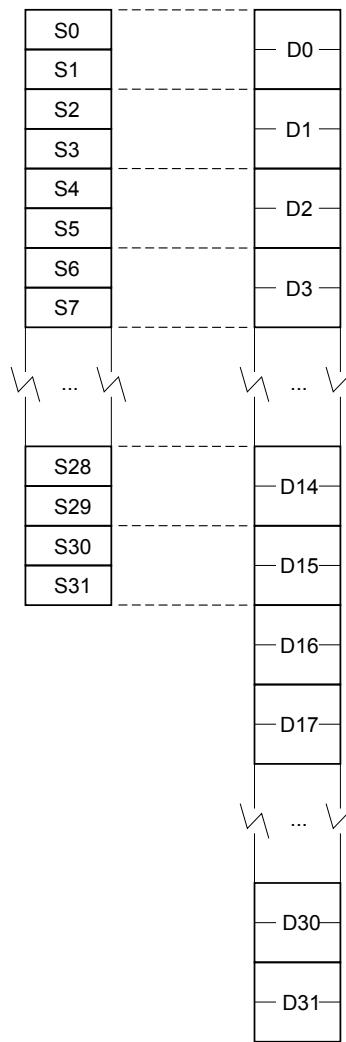
In AArch64 state, the register bank includes thirty-two 128-bit registers and has a new register packing model.

Floating point instructions in A64 are closely based on VFPv4 and A32, but with new instruction mnemonics and some functional enhancements.

## B2.2 Extension register bank mapping for floating-point in AArch32 state

The floating-point extension register bank is a collection of registers that can be accessed as either 32-bit or 64-bit registers. It is distinct from the Arm core register bank.

The following figure shows the views of the extension register bank, and the overlap between the different size registers. For example, the 64-bit register D0 is an alias for two consecutive 32-bit registers S0 and S1. The 64-bit registers D16 and D17 do not have an alias.



**Figure B2-1 Extension register bank for floating-point in AArch32 state**

The aliased views enable half-precision, single-precision, and double-precision values to coexist in different non-overlapped registers at the same time.

You can also use the same overlapped registers to store half-precision, single-precision, and double-precision values at different times.

Do not attempt to use overlapped 32-bit and 64-bit registers at the same time because it creates meaningless results.

The mapping between the registers is as follows:

- S<sub><2n></sub> maps to the least significant half of D<sub><n></sub>
- S<sub><2n+1></sub> maps to the most significant half of D<sub><n></sub>

For example, you can access the least significant half of register D6 by referring to S12, and the most significant half of D6 by referring to S13.

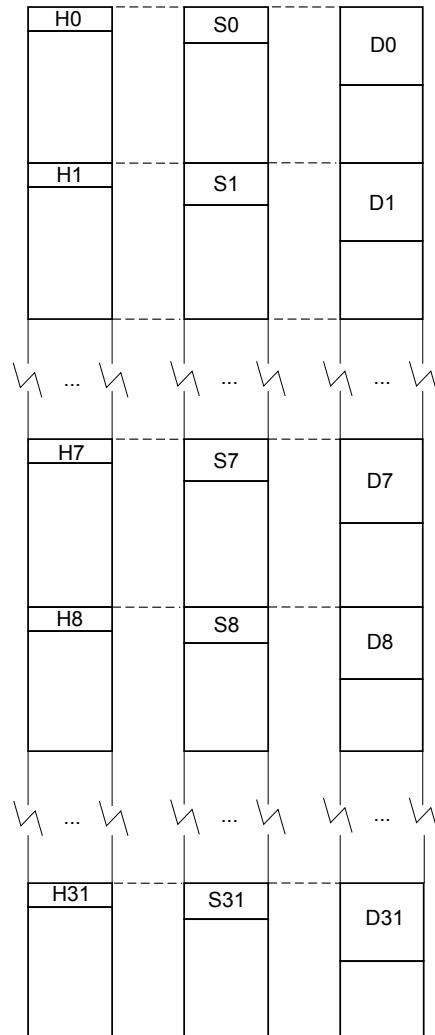
**Related concepts**

[B2.4 Views of the floating-point extension register bank in AArch32 state on page B2-118](#)

## B2.3 Extension register bank mapping in AArch64 state

The extension register bank is a collection of registers that can be accessed as 16-bit, 32-bit, or 64-bit. It is distinct from the Arm core register bank.

The following figure shows the views of the extension register bank, and the overlap between the different size registers.



**Figure B2-2 Extension register bank for floating-point in AArch64 state**

The mapping between the registers is as follows:

- $S_{<n>}$  maps to the least significant half of  $D_{<n>}$
- $H_{<n>}$  maps to the least significant half of  $S_{<n>}$

For example, you can access the least significant half of register  $D7$  by referring to  $S7$ .

### Related concepts

[B2.5 Views of the floating-point extension register bank in AArch64 state](#) on page B2-119

## B2.4 Views of the floating-point extension register bank in AArch32 state

Floating-point can have different views of the extension register bank in AArch32 state.

The floating-point extension register bank can be viewed as:

- Thirty-two 64-bit registers, D0-D31.
- Thirty-two 32-bit registers, S0-S31. Only half of the register bank is accessible in this view.
- A combination of registers from these views.

64-bit floating-point registers are called double-precision registers and can contain double-precision floating-point values. 32-bit floating-point registers are called single-precision registers and can contain either a single-precision or two half-precision floating-point values.

### ***Related concepts***

[B2.2 Extension register bank mapping for floating-point in AArch32 state](#) on page B2-115

## B2.5 Views of the floating-point extension register bank in AArch64 state

Floating-point can have different views of the extension register bank in AArch64 state.

The floating-point extension register bank can be viewed as:

- Thirty-two 64-bit registers D0-D31.
- Thirty-two 32-bit registers S0-S31.
- Thirty-two 16-bit registers H0-H31.
- A combination of registers from these views.

### ***Related concepts***

[B2.3 Extension register bank mapping in AArch64 state](#) on page B2-117

## B2.6 Differences between A32/T32 and A64 floating-point instruction syntax

The syntax and mnemonics of A64 floating-point instructions are based on those in A32/T32 but with some differences.

The following table describes the main differences.

**Table B2-1 Differences in syntax and mnemonics between A32/T32 and A64 floating-point instructions**

A32/T32	A64
All floating-point instruction mnemonics begin with V, for example VMAX.	The first letter of the instruction mnemonic indicates the data type of the instruction. For example, SMAX, UMAX, and FMAX mean signed, unsigned, and floating-point respectively. No suffix means the type is irrelevant and P means polynomial.
A mnemonic qualifier specifies the type and width of elements in a vector. For example, in the following instruction, U32 means 32-bit unsigned integers:  VMAX.U32 Q0, Q1, Q2	A register qualifier specifies the data width and the number of elements in the register. For example, in the following instruction .4S means 4 32-bit elements:  UMAX V0.4S, V1.4S, V2.4S
You can append a condition code to most floating-point instruction mnemonics to make them conditional.	A64 has no conditionally executed floating-point instructions.
The floating-point select instruction, VSEL, is unconditionally executed but uses a condition code as an operand. You append the condition code to the mnemonic, for example:  VSELEQ.F32 S1,S2,S3	There are several floating-point instructions that use a condition code as an operand. You specify the condition code in the final operand position, for example:  FCSEL S1,S2,S3,EQ
The P mnemonic qualifier which indicates pairwise instructions is a prefix, for example, VPADD.	The P mnemonic qualifier is a suffix, for example ADDP.

## B2.7 Load values to floating-point registers

To load a register with a floating-point immediate value, use VMOV in A32 or FMOV in A64. Both instructions exist in scalar and vector forms.

### ***Related reference***

*VLDR pseudo-instruction (floating-point)*

*C4.22 VMOV (floating-point) on page C4-623*

*D4.32 FMOV (scalar, immediate) on page D4-1067*

## B2.8 Conditional execution of A32/T32 floating-point instructions

You can execute floating-point instructions conditionally, in the same way as most A32 and T32 instructions.

You cannot use any of the following floating-point instructions in an IT block:

- VRINT{A, N, P, M}.
- VSEL.
- VCVT{A, N, P, M}.
- VMAXNM.
- VMINNM.

In addition, specifying any other floating-point instruction in an IT block is deprecated.

Most A32 floating-point instructions can be conditionally executed, by appending a condition code suffix to the instruction.

### ***Related concepts***

[C1.2 Conditional execution in A32 code](#) on page C1-135

[C1.3 Conditional execution in T32 code](#) on page C1-136

### ***Related reference***

[C1.11 Comparison of condition code meanings in integer and floating-point code](#) on page C1-144

[C1.9 Condition code suffixes](#) on page C1-142

## B2.9 Floating-point exceptions for floating-point in A32/T32 instructions

The floating-point extension records floating-point exceptions in the FPSCR cumulative flags.

It records the following exceptions:

### Invalid operation

The exception is caused if the result of an operation has no mathematical value or cannot be represented.

### Division by zero

The exception is caused if a divide operation has a zero divisor and a dividend that is not zero, an infinity or a NaN.

### Overflow

The exception is caused if the absolute value of the result of an operation, produced after rounding, is greater than the maximum positive normalized number for the destination precision.

### Underflow

The exception is caused if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, and the rounded result is inexact.

### Inexact

The exception is caused if the result of an operation is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

### Input denormal

The exception is caused if a denormalized input operand is replaced in the computation by a zero.

The descriptions of the floating-point instructions that can cause floating-point exceptions include a subsection listing the exceptions. If there is no such subsection, that instruction cannot cause any floating-point exception.

### Related concepts

[B2.13 Flush-to-zero mode in floating-point](#) on page B2-127

### Related reference

[Chapter C4 Floating-point Instructions \(32-bit\)](#) on page C4-599

### Related information

[Arm Architecture Reference Manual](#)

## B2.10 Floating-point data types in A32/T32 instructions

Most floating-point instructions use a data type specifier to define the size and type of data that the instruction operates on.

Data type specifiers in floating-point instructions consist of a letter indicating the type of data, usually followed by a number indicating the width. They are separated from the instruction mnemonic by a point.

The following data types are available in floating-point instructions:

### 16-bit

F16

### 32-bit

F32 (or F)

### 64-bit

F64 (or D)

The datatype of the second (or only) operand is specified in the instruction.

---

— Note —

- Most instructions have a restricted range of permitted data types. See the instruction descriptions for details. However, the data type description is flexible:
    - If the description specifies I, you can also use the S or U data types.
    - If only the data size is specified, you can specify a type (S, U, P or F).
    - If no data type is specified, you can specify a data type.
- 

### Related concepts

[B1.11 Polynomial arithmetic over {0,1} on page B1-101](#)

## B2.11 Extended notation extension for floating-point in A32/T32 code

`armasm` implements an extension to the architectural floating-point assembly syntax, called *extended notation*. This extension allows you to include datatype information or scalar indexes in register names.

————— Note —————

Extended notation is not supported for A64 code.

If you use extended notation, you do not have to include the data type or scalar index information in every instruction.

Register names can be any of the following:

**Untyped**

The register name specifies the register, but not what datatype it contains, nor any index to a particular scalar within the register.

**Untyped with scalar index**

The register name specifies the register, but not what datatype it contains, It specifies an index to a particular scalar within the register.

**Typed**

The register name specifies the register, and what datatype it contains, but not any index to a particular scalar within the register.

**Typed with scalar index**

The register name specifies the register, what datatype it contains, and an index to a particular scalar within the register.

Use the `SN` and `DN` directives to define names for typed and scalar registers.

**Related concepts**

[B2.10 Floating-point data types in A32/T32 instructions](#) on page B2-124

## B2.12 Floating-point system registers in AArch32 state

Floating-point system registers are accessible in all implementations of floating-point.

For exception levels using AArch32, the following floating-point system registers are accessible in all floating-point implementations:

- FPSCR, the floating-point status and control register.
- FPEXC, the floating-point exception register.
- FPSID, the floating-point system ID register.

A particular floating-point implementation can have additional registers. For more information, see the Technical Reference Manual for your processor.

### *Related information*

*Arm Architecture Reference Manual*

## B2.13 Flush-to-zero mode in floating-point

Flush-to-zero mode replaces denormalized numbers with zero. This does not comply with IEEE 754 arithmetic, but in some circumstances can improve performance considerably.

Some implementations of floating-point use support code to handle denormalized numbers. The performance of such systems, in calculations involving denormalized numbers, is much less than it is in normal calculations.

Flush-to-zero mode in floating-point always preserves the sign bit.

### ***Related concepts***

[B2.15 The effects of using flush-to-zero mode in floating-point](#) on page B2-129

### ***Related reference***

[B2.14 When to use flush-to-zero mode in floating-point](#) on page B2-128

[B2.16 Floating-point operations not affected by flush-to-zero mode](#) on page B2-130

## B2.14 When to use flush-to-zero mode in floating-point

You can change between flush-to-zero mode and normal mode, depending on the requirements of different parts of your code.

You must select flush-to-zero mode if all the following are true:

- IEEE 754 compliance is not a requirement for your system.
- The algorithms you are using sometimes generate denormalized numbers.
- Your system uses support code to handle denormalized numbers.
- The algorithms you are using do not depend for their accuracy on the preservation of denormalized numbers.
- The algorithms you are using do not generate frequent exceptions as a result of replacing denormalized numbers with 0.

You select flush-to-zero mode in one of the following ways:

- In A32 code, by setting the FZ bit in the FPSCR to 1. You do this using the VMRS and VMSR instructions.
- In A64 code, by setting the FZ bit in the FPCR to 1. You do this using the MRS and MSR instructions.

You can change between flush-to-zero and normal mode at any time, if different parts of your code have different requirements. Numbers already in registers are not affected by changing mode.

### ***Related concepts***

[B2.13 Flush-to-zero mode in floating-point on page B2-127](#)

[B2.15 The effects of using flush-to-zero mode in floating-point on page B2-129](#)

## B2.15 The effects of using flush-to-zero mode in floating-point

In flush-to-zero mode, denormalized inputs are treated as zero. Results that are too small to be represented in a normalized number are replaced with zero.

With certain exceptions, flush-to-zero mode has the following effects on floating-point operations:

- A denormalized number is treated as 0 when used as an input to a floating-point operation. The source register is not altered.
- If the result of a single-precision floating-point operation, before rounding, is in the range  $-2^{-126}$  to  $+2^{-126}$ , it is replaced by 0.
- If the result of a double-precision floating-point operation, before rounding, is in the range  $-2^{-1022}$  to  $+2^{-1022}$ , it is replaced by 0.

In flush-to-zero mode, an Input Denormal exception occurs whenever a denormalized number is used as an operand. An Underflow exception occurs when a result is flushed-to-zero.

### *Related concepts*

[B2.13 Flush-to-zero mode in floating-point](#) on page B2-127

### *Related reference*

[B2.16 Floating-point operations not affected by flush-to-zero mode](#) on page B2-130

## B2.16 Floating-point operations not affected by flush-to-zero mode

Some floating-point instructions can be carried out on denormalized numbers even in flush-to-zero mode, without flushing the results to zero.

These instructions are as follows:

- Absolute value and negate (VABS and VNEG).
- Load and store (VLDR and VSTR).
- Load multiple and store multiple (VLDM and VSTM).
- Transfer between extension registers and general-purpose registers (VMOV).

### *Related concepts*

[B2.13 Flush-to-zero mode in floating-point](#) on page B2-127

### *Related reference*

[C4.2 VABS \(floating-point\)](#) on page C4-603

[C4.14 VLDM \(floating-point\)](#) on page C4-615

[C4.15 VLDR \(floating-point\)](#) on page C4-616

[C4.38 VSTM \(floating-point\)](#) on page C4-639

[C4.39 VSTR \(floating-point\)](#) on page C4-640

[C3.54 VLDM](#) on page C3-503

[C3.55 VLDR](#) on page C3-504

[C3.131 VSTM](#) on page C3-580

[C3.134 VSTR](#) on page C3-585

[C4.23 VMOV \(between one general-purpose register and single precision floating-point register\)](#)  
on page C4-624

[C3.70 VMOV \(between two general-purpose registers and a 64-bit extension register\)](#) on page C3-519

[C4.29 VNEG \(floating-point\)](#) on page C4-630

[C3.83 VNEG](#) on page C3-532

## **Part C**

# **A32/T32 Instruction Set Reference**



# Chapter C1

## Condition Codes

Describes condition codes and conditional execution of A32 and T32 code.

It contains the following sections:

- [\*C1.1 Conditional instructions\* on page C1-134.](#)
- [\*C1.2 Conditional execution in A32 code\* on page C1-135.](#)
- [\*C1.3 Conditional execution in T32 code\* on page C1-136.](#)
- [\*C1.4 Condition flags\* on page C1-137.](#)
- [\*C1.5 Updates to the condition flags in A32/T32 code\* on page C1-138.](#)
- [\*C1.6 Floating-point instructions that update the condition flags\* on page C1-139.](#)
- [\*C1.7 Carry flag\* on page C1-140.](#)
- [\*C1.8 Overflow flag\* on page C1-141.](#)
- [\*C1.9 Condition code suffixes\* on page C1-142.](#)
- [\*C1.10 Condition code suffixes and related flags\* on page C1-143.](#)
- [\*C1.11 Comparison of condition code meanings in integer and floating-point code\* on page C1-144.](#)
- [\*C1.12 Benefits of using conditional execution in A32 and T32 code\* on page C1-146.](#)
- [\*C1.13 Example showing the benefits of conditional instructions in A32 and T32 code\* on page C1-147.](#)
- [\*C1.14 Optimization for execution speed\* on page C1-150.](#)

## C1.1 Conditional instructions

A32 and T32 instructions can execute conditionally on the condition flags set by a previous instruction.

The conditional instruction can occur either:

- Immediately after the instruction that updated the flags.
- After any number of intervening instructions that have not updated the flags.

In AArch32 state, whether an instruction can be conditional or not depends on the instruction set state that the processor is in.

To make an instruction conditional, you must add a condition code suffix to the instruction mnemonic. The condition code suffix enables the processor to test a condition based on the flags. If the condition test of a conditional instruction fails, the instruction:

- Does not execute.
- Does not write any value to its destination register.
- Does not affect any of the flags.
- Does not generate any exception.

### ***Related concepts***

[C1.2 Conditional execution in A32 code](#) on page C1-135

[C1.3 Conditional execution in T32 code](#) on page C1-136

### ***Related reference***

[C1.10 Condition code suffixes and related flags](#) on page C1-143

[C1.5 Updates to the condition flags in A32/T32 code](#) on page C1-138

## C1.2 Conditional execution in A32 code

Almost all A32 instructions can be executed conditionally on the value of the condition flags in the APSR. You can either add a condition code suffix to the instruction or you can conditionally skip over the instruction using a conditional branch instruction.

Using conditional branch instructions to control the flow of execution can be more efficient when a series of instructions depend on the same condition.

### Conditional instructions to control execution

```
; flags set by a previous instruction
    LSLEQ r0, r0, #24
    ADDEQ r0, r0, #2
;...
```

### Conditional branch to control execution

```
; flags set by a previous instruction
    BNE over
    LSL r0, r0, #24
    ADD r0, r0, #2
over
;...
```

#### Related concepts

[C1.3 Conditional execution in T32 code](#) on page C1-136

## C1.3 Conditional execution in T32 code

In T32 code, there are several ways to achieve conditional execution. You can conditionally skip over the instruction using a conditional branch instruction.

Instructions can also be conditionally executed by using either of the following:

- CBZ and CBNZ.
- The IT (If-Then) instruction.

The T32 CBZ (Conditional Branch on Zero) and CBNZ (Conditional Branch on Non-Zero) instructions compare the value of a register against zero and branch on the result.

IT is a 16-bit instruction that enables a single subsequent 16-bit T32 instruction from a restricted set to be conditionally executed, based on the value of the condition flags, and the condition code suffix specified.

### Conditional instructions using IT block

```
; flags set by a previous instruction
    IT EQ
    LSLEQ r0, r0, #24
;...
```

The use of the IT instruction is deprecated when any of the following are true:

- There is more than one instruction in the IT block.
- There is a 32-bit instruction in the IT block.
- The instruction in the IT block references the PC.

#### Related concepts

[C1.2 Conditional execution in A32 code](#) on page C1-135

#### Related reference

[C2.44 IT](#) on page C2-222

[C2.23 CBZ and CBNZ](#) on page C2-195

## C1.4 Condition flags

The N, Z, C, and V condition flags are held in the APSR.

The condition flags are held in the APSR. They are set or cleared as follows:

**N**

Set to 1 when the result of the operation is negative, cleared to 0 otherwise.

**Z**

Set to 1 when the result of the operation is zero, cleared to 0 otherwise.

**C**

Set to 1 when the operation results in a carry, or when a subtraction results in no borrow, cleared to 0 otherwise.

**V**

Set to 1 when the operation causes overflow, cleared to 0 otherwise.

C is set in one of the following ways:

- For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-addition/subtractions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.

Overflow occurs if the result of a signed add, subtract, or compare is greater than or equal to  $2^{31}$ , or less than  $-2^{31}$ .

### **Related reference**

[C1.5 Updates to the condition flags in A32/T32 code on page C1-138](#)

[C1.10 Condition code suffixes and related flags on page C1-143](#)

[D1.3 Updates to the condition flags in A64 code on page D1-650](#)

[D1.8 Condition code suffixes and related flags on page D1-655](#)

## C1.5 Updates to the condition flags in A32/T32 code

In AArch32 state, the condition flags are held in the *Application Program Status Register* (APSR). You can read and modify the flags using the read-modify-write procedure.

Most A32 and T32 data processing instructions have an option to update the condition flags according to the result of the operation. Instructions with the optional S suffix update the flags. Conditional instructions that are not executed have no effect on the flags.

Which flags are updated depends on the instruction. Some instructions update all flags, and some update a subset of the flags. If a flag is not updated, the original value is preserved. The description of each instruction mentions the effect that it has on the flags.

————— **Note** ———

Most instructions update the condition flags only if the S suffix is specified. The instructions **CMP**, **CMN**, **TEQ**, and **TST** always update the flags.

**Related concepts**

[C1.1 Conditional instructions](#) on page C1-134

**Related reference**

[C1.4 Condition flags](#) on page C1-137

[D1.3 Updates to the condition flags in A64 code](#) on page D1-650

[C1.10 Condition code suffixes and related flags](#) on page C1-143

[Chapter C2 A32 and T32 Instructions](#) on page C2-151

## C1.6 Floating-point instructions that update the condition flags

The only A32/T32 floating-point instructions that can update the condition flags are VCMP and VCMPE. Other floating-point or Advanced SIMD instructions cannot modify the flags.

VCMP and VCMPE do not update the flags directly, but update a separate set of flags in the *Floating-Point Status and Control Register* (FPSCR). To use these flags to control conditional instructions, including conditional floating-point instructions, you must first update the condition flags yourself. To do this, copy the flags from the FPSCR into the APSR using a VMRS instruction:

```
VMRS APSR_nzcv, FPSCR
```

### Related concepts

[C1.7 Carry flag](#) on page C1-140

[C1.8 Overflow flag](#) on page C1-141

### Related reference

[D1.3 Updates to the condition flags in A64 code](#) on page D1-650

[C4.4 VCMP, VCMPE](#) on page C4-605

[C3.75 VMRS](#) on page C3-524

[C4.26 VMRS \(floating-point\)](#) on page C4-627

### Related information

[Arm Architecture Reference Manual](#)

## C1.7 Carry flag

The carry (C) flag is set when an operation results in a carry, or when a subtraction results in no borrow.

In A32/T32 code, C is set in one of the following ways:

- For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-additions/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-additions/subtractions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.
- The floating-point compare instructions, VCMP and VCMPE set the C flag and the other condition flags in the FPSCR to the result of the comparison.

### ***Related concepts***

[C1.8 Overflow flag on page C1-141](#)

### ***Related reference***

[A2.7 Predeclared core register names in AArch32 state on page A2-61](#)

[A3.5 Predeclared core register names in AArch64 state on page A3-76](#)

[C1.10 Condition code suffixes and related flags on page C1-143](#)

[C1.5 Updates to the condition flags in A32/T32 code on page C1-138](#)

[D1.3 Updates to the condition flags in A64 code on page D1-650](#)

## C1.8 Overflow flag

Overflow can occur for add, subtract, and compare operations.

In A32/T32 code, overflow occurs if the result of the operation is greater than or equal to  $2^{31}$ , or less than  $-2^{31}$ .

### ***Related concepts***

[C1.7 Carry flag on page C1-140](#)

### ***Related reference***

[A2.7 Predeclared core register names in AArch32 state on page A2-61](#)

[C1.5 Updates to the condition flags in A32/T32 code on page C1-138](#)

[D1.3 Updates to the condition flags in A64 code on page D1-650](#)

## C1.9 Condition code suffixes

Instructions that can be conditional have an optional two character condition code suffix.

Condition codes are shown in syntax descriptions as {cond}. The following table shows the condition codes that you can use:

Table C1-1 Condition code suffixes

Suffix	Meaning
EQ	Equal
NE	Not equal
CS	Carry set (identical to HS)
HS	Unsigned higher or same (identical to CS)
CC	Carry clear (identical to LO)
LO	Unsigned lower (identical to CC)
MI	Minus or negative result
PL	Positive or zero result
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (this is the default)

### Note

The meaning of some of these condition codes depends on whether the instruction that last updated the condition flags is a floating-point or integer instruction.

### Related reference

[C1.11 Comparison of condition code meanings in integer and floating-point code](#) on page C1-144

[C2.44 IT](#) on page C2-222

[C3.75 VMRS](#) on page C3-524

[C4.26 VMRS \(floating-point\)](#) on page C4-627

## C1.10 Condition code suffixes and related flags

Condition code suffixes define the conditions that must be met for the instruction to execute.

The following table shows the condition codes that you can use and the flag settings they depend on:

**Table C1-2 Condition code suffixes and related flags**

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned $\geq$ )
CC or LO	C clear	Lower (unsigned $<$ )
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned $>$ )
LS	C clear or Z set	Lower or same (unsigned $\leq$ )
GE	N and V the same	Signed $\geq$
LT	N and V differ	Signed $<$
GT	Z clear, N and V the same	Signed $>$
LE	Z set, N and V differ	Signed $\leq$
AL	Any	Always. This suffix is normally omitted.

The optional condition code is shown in syntax descriptions as {cond}. This condition is encoded in A32 instructions. For T32 instructions, the condition is encoded in a preceding IT instruction. An instruction with a condition code is only executed if the condition flags meet the specified condition.

The following is an example of conditional execution in A32 code:

```

ADD      r0, r1, r2      ; r0 = r1 + r2, don't update flags
ADDS     r0, r1, r2      ; r0 = r1 + r2, and update flags
ADDSCS   r0, r1, r2      ; If C flag set then r0 = r1 + r2,
; and update flags
CMP      r0, r1          ; update flags based on r0-r1.

```

### Related concepts

[C1.1 Conditional instructions](#) on page C1-134

### Related reference

[C1.4 Condition flags](#) on page C1-137

[C1.11 Comparison of condition code meanings in integer and floating-point code](#) on page C1-144

[C1.5 Updates to the condition flags in A32/T32 code](#) on page C1-138

[D1.3 Updates to the condition flags in A64 code](#) on page D1-650

[Chapter C2 A32 and T32 Instructions](#) on page C2-151

## C1.11 Comparison of condition code meanings in integer and floating-point code

The meaning of the condition code mnemonic suffixes depends on whether the condition flags were set by a floating-point instruction or by an A32 or T32 data processing instruction.

This is because:

- Floating-point values are never unsigned, so the unsigned conditions are not required.
- Not-a-Number (NaN) values have no ordering relationship with numbers or with each other, so additional conditions are required to account for unordered results.

The meaning of the condition code mnemonic suffixes is shown in the following table:

**Table C1-3 Condition codes**

Suffix	Meaning after integer data processing instruction	Meaning after floating-point instruction
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS	Carry set	Greater than or equal, or unordered
HS	Unsigned higher or same	Greater than or equal, or unordered
CC	Carry clear	Less than
LO	Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

---

**Note**

---

The type of the instruction that last updated the condition flags determines the meaning of the condition codes.

---

**Related concepts**

[C1.1 Conditional instructions](#) on page C1-134

**Related reference**

[C1.10 Condition code suffixes and related flags](#) on page C1-143

[C1.5 Updates to the condition flags in A32/T32 code](#) on page C1-138

[D1.3 Updates to the condition flags in A64 code](#) on page D1-650

[C4.4 VCMP, VCMPE](#) on page C4-605

[C3.75 VMRS](#) on page C3-524

*C4.26 VMRS (floating-point) on page C4-627*

**Related information**

*Arm Architecture Reference Manual*

## C1.12 Benefits of using conditional execution in A32 and T32 code

It can be more efficient to use conditional instructions rather than conditional branches.

You can use conditional execution of A32 instructions to reduce the number of branch instructions in your code, and improve code density. The IT instruction in T32 achieves a similar improvement.

Branch instructions are also expensive in processor cycles. On Arm processors without branch prediction hardware, it typically takes three processor cycles to refill the processor pipeline each time a branch is taken.

Some Arm processors have branch prediction hardware. In systems using these processors, the pipeline only has to be flushed and refilled when there is a misprediction.

### ***Related concepts***

*C1.13 Example showing the benefits of conditional instructions in A32 and T32 code on page C1-147*

## C1.13 Example showing the benefits of conditional instructions in A32 and T32 code

Using conditional instructions rather than conditional branches can save both code size and cycles.

This example shows the difference between using branches and using conditional instructions. It uses the Euclid algorithm for the *Greatest Common Divisor* (gcd) to show how conditional instructions improve code size and speed.

In C the gcd algorithm can be expressed as:

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

The following examples show implementations of the gcd algorithm with and without conditional instructions.

### Example of conditional execution using branches in A32 code

This example is an A32 code implementation of the gcd algorithm. It achieves conditional execution by using conditional branches, rather than individual conditional instructions:

gcd	CMP	r0, r1	
	BEQ	end	
	BLT	less	
	SUBS	r0, r0, r1 ; could be SUB r0, r0, r1 for A32	
	B	gcd	
less	SUBS	r1, r1, r0 ; could be SUB r1, r1, r0 for A32	
	B	gcd	
	end		

The code is seven instructions long because of the number of branches. Every time a branch is taken, the processor must refill the pipeline and continue from the new location. The other instructions and non-executed branches use a single cycle each.

The following table shows the number of cycles this implementation uses on an Arm7™ processor when R0 equals 1 and R1 equals 2.

**Table C1-4 Conditional branches only**

R0: a	R1: b	Instruction	Cycles (Arm7)
1	2	CMP r0, r1	1
1	2	BEQ end	1 (not executed)
1	2	BLT less	3
1	2	SUB r1, r1, r0	1
1	2	B gcd	3
1	1	CMP r0, r1	1
1	1	BEQ end	3
			Total = 13

### Example of conditional execution using conditional instructions in A32 code

This example is an A32 code implementation of the gcd algorithm using individual conditional instructions in A32 code. The gcd algorithm only takes four instructions:

```
gcd
    CMP    r0, r1
    SUBGT r0, r0, r1
    SUBLE r1, r1, r0
    BNE    gcd
```

In addition to improving code size, in most cases this code executes faster than the version that uses only branches.

The following table shows the number of cycles this implementation uses on an Arm7 processor when R0 equals 1 and R1 equals 2.

**Table C1-5 All instructions conditional**

R0: a	R1: b	Instruction	Cycles (Arm7)
1	2	CMP r0, r1	1
1	2	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1
1	1	BNE gcd	3
1	1	CMP r0,r1	1
1	1	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1 (not executed)
1	1	BNE gcd	1 (not executed)
			Total = 10

Comparing this with the example that uses only branches:

- Replacing branches with conditional execution of all instructions saves three cycles.
- Where R0 equals R1, both implementations execute in the same number of cycles. For all other cases, the implementation that uses conditional instructions executes in fewer cycles than the implementation that uses branches only.

### Example of conditional execution using conditional instructions in T32 code

You can use the IT instruction to write conditional instructions in T32 code. The T32 code implementation of the gcd algorithm using conditional instructions is similar to the implementation in A32 code. The implementation in T32 code is:

```
gcd
    CMP    r0, r1
    ITE    GT
    SUBGT r0, r0, r1
    SUBLE r1, r1, r0
    BNE    gcd
```

These instructions assemble equally well to A32 or T32 code. The assembler checks the IT instructions, but omits them on assembly to A32 code.

It requires one more instruction in T32 code (the IT instruction) than in A32 code, but the overall code size is 10 bytes in T32 code, compared with 16 bytes in A32 code.

### Example of conditional execution code using branches in T32 code

In architectures before Armv6T2, there is no IT instruction and therefore T32 instructions cannot be executed conditionally except for the B branch instruction. The gcd algorithm must be written with

conditional branches and is similar to the A32 code implementation using branches, without conditional instructions.

The T32 code implementation of the gcd algorithm without conditional instructions requires seven instructions. The overall code size is 14 bytes. This figure is even less than the A32 implementation that uses conditional instructions, which uses 16 bytes.

In addition, on a system using 16-bit memory this T32 implementation runs faster than both A32 implementations because only one memory access is required for each 16-bit T32 instruction, whereas each 32-bit A32 instruction requires two fetches.

#### **Related concepts**

[C1.12 Benefits of using conditional execution in A32 and T32 code](#) on page C1-146

[C1.14 Optimization for execution speed](#) on page C1-150

#### **Related reference**

[C2.44 IT](#) on page C2-222

[C1.10 Condition code suffixes and related flags](#) on page C1-143

#### **Related information**

[Arm Architecture Reference Manual](#)

## C1.14 Optimization for execution speed

To optimize code for execution speed you must have detailed knowledge of the instruction timings, branch prediction logic, and cache behavior of your target system.

For more information, see the Technical Reference Manual for your processor.

### ***Related information***

*Arm Architecture Reference Manual*

*Further reading*

# Chapter C2

## A32 and T32 Instructions

Describes the A32 and T32 instructions supported in AArch32 state.

It contains the following sections:

- [\*C2.1 A32 and T32 instruction summary\*](#) on page C2-156.
- [\*C2.2 Instruction width specifiers\*](#) on page C2-161.
- [\*C2.3 Flexible second operand \(Operand2\)\*](#) on page C2-162.
- [\*C2.4 Syntax of Operand2 as a constant\*](#) on page C2-163.
- [\*C2.5 Syntax of Operand2 as a register with optional shift\*](#) on page C2-164.
- [\*C2.6 Shift operations\*](#) on page C2-165.
- [\*C2.7 Saturating instructions\*](#) on page C2-168.
- [\*C2.8 ADC\*](#) on page C2-169.
- [\*C2.9 ADD\*](#) on page C2-171.
- [\*C2.10 ADR \(PC-relative\)\*](#) on page C2-174.
- [\*C2.11 ADR \(register-relative\)\*](#) on page C2-176.
- [\*C2.12 AND\*](#) on page C2-178.
- [\*C2.13 ASR\*](#) on page C2-180.
- [\*C2.14 B\*](#) on page C2-182.
- [\*C2.15 BFC\*](#) on page C2-184.
- [\*C2.16 BFI\*](#) on page C2-185.
- [\*C2.17 BIC\*](#) on page C2-186.
- [\*C2.18 BKPT\*](#) on page C2-188.
- [\*C2.19 BL\*](#) on page C2-189.
- [\*C2.20 BLX, BLXNS\*](#) on page C2-190.
- [\*C2.21 BX, BXNS\*](#) on page C2-192.
- [\*C2.22 BXJ\*](#) on page C2-194.
- [\*C2.23 CBZ and CBNZ\*](#) on page C2-195.

- C2.24 *CDP and CDP2* on page C2-196.
- C2.25 *CLREX* on page C2-197.
- C2.26 *CLZ* on page C2-198.
- C2.27 *CMP and CMN* on page C2-199.
- C2.28 *CPS* on page C2-201.
- C2.29 *CRC32* on page C2-203.
- C2.30 *CRC32C* on page C2-204.
- C2.31 *CSDB* on page C2-205.
- C2.32 *DBG* on page C2-207.
- C2.33 *DCPS1 (T32 instruction)* on page C2-208.
- C2.34 *DCPS2 (T32 instruction)* on page C2-209.
- C2.35 *DCPS3 (T32 instruction)* on page C2-210.
- C2.36 *DMB* on page C2-211.
- C2.37 *DSB* on page C2-213.
- C2.38 *EOR* on page C2-215.
- C2.39 *ERET* on page C2-217.
- C2.40 *ESB* on page C2-218.
- C2.41 *HLT* on page C2-219.
- C2.42 *HVC* on page C2-220.
- C2.43 *ISB* on page C2-221.
- C2.44 *IT* on page C2-222.
- C2.45 *LDA* on page C2-225.
- C2.46 *LDAEX* on page C2-226.
- C2.47 *LDC and LDC2* on page C2-228.
- C2.48 *LDM* on page C2-230.
- C2.49 *LDR (immediate offset)* on page C2-232.
- C2.50 *LDR (PC-relative)* on page C2-234.
- C2.51 *LDR (register offset)* on page C2-236.
- C2.52 *LDR (register-relative)* on page C2-238.
- C2.53 *LDR, unprivileged* on page C2-240.
- C2.54 *LDREX* on page C2-242.
- C2.55 *LSL* on page C2-244.
- C2.56 *LSR* on page C2-246.
- C2.57 *MCR and MCR2* on page C2-248.
- C2.58 *MCRR and MCRR2* on page C2-249.
- C2.59 *MLA* on page C2-250.
- C2.60 *MLS* on page C2-251.
- C2.61 *MOV* on page C2-252.
- C2.62 *MOVT* on page C2-254.
- C2.63 *MRC and MRC2* on page C2-255.
- C2.64 *MRRC and MRRC2* on page C2-256.
- C2.65 *MRS (PSR to general-purpose register)* on page C2-257.
- C2.66 *MRS (system coprocessor register to general-purpose register)* on page C2-259.
- C2.67 *MSR (general-purpose register to system coprocessor register)* on page C2-260.
- C2.68 *MSR (general-purpose register to PSR)* on page C2-261.
- C2.69 *MUL* on page C2-263.
- C2.70 *MVN* on page C2-264.
- C2.71 *NOP* on page C2-266.
- C2.72 *ORN (T32 only)* on page C2-267.
- C2.73 *ORR* on page C2-268.
- C2.74 *PKHBT and PKHTB* on page C2-270.
- C2.75 *PLD, PLDW, and PLI* on page C2-272.
- C2.76 *POP* on page C2-274.
- C2.77 *PUSH* on page C2-275.
- C2.78 *QADD* on page C2-276.
- C2.79 *QADD8* on page C2-277.

- C2.80 *QADD16* on page C2-278.
- C2.81 *QASX* on page C2-279.
- C2.82 *QDADD* on page C2-280.
- C2.83 *QDSUB* on page C2-281.
- C2.84 *QSAX* on page C2-282.
- C2.85 *QSUB* on page C2-283.
- C2.86 *QSUB8* on page C2-284.
- C2.87 *QSUB16* on page C2-285.
- C2.88 *RBIT* on page C2-286.
- C2.89 *REV* on page C2-287.
- C2.90 *REV16* on page C2-288.
- C2.91 *REVSH* on page C2-289.
- C2.92 *RFE* on page C2-290.
- C2.93 *ROR* on page C2-292.
- C2.94 *RRX* on page C2-294.
- C2.95 *RSB* on page C2-296.
- C2.96 *RSC* on page C2-298.
- C2.97 *SADD8* on page C2-300.
- C2.98 *SADD16* on page C2-302.
- C2.99 *SASX* on page C2-304.
- C2.100 *SBC* on page C2-306.
- C2.101 *SBFX* on page C2-308.
- C2.102 *SDIV* on page C2-309.
- C2.103 *SEL* on page C2-310.
- C2.104 *SETEND* on page C2-312.
- C2.105 *SETPAN* on page C2-313.
- C2.106 *SEV* on page C2-314.
- C2.107 *SEVL* on page C2-315.
- C2.108 *SG* on page C2-316.
- C2.109 *SHADD8* on page C2-317.
- C2.110 *SHADD16* on page C2-318.
- C2.111 *SHASX* on page C2-319.
- C2.112 *SHSAX* on page C2-320.
- C2.113 *SHSUB8* on page C2-321.
- C2.114 *SHSUB16* on page C2-322.
- C2.115 *SMC* on page C2-323.
- C2.116 *SMLAxy* on page C2-324.
- C2.117 *SMLAD* on page C2-326.
- C2.118 *SMLAL* on page C2-327.
- C2.119 *SMLALD* on page C2-328.
- C2.120 *SMLALxy* on page C2-329.
- C2.121 *SMLAWy* on page C2-331.
- C2.122 *SMLSD* on page C2-332.
- C2.123 *SMLSxD* on page C2-333.
- C2.124 *SMMLA* on page C2-334.
- C2.125 *SMMLS* on page C2-335.
- C2.126 *SMMUL* on page C2-336.
- C2.127 *SMUAD* on page C2-337.
- C2.128 *SMULxy* on page C2-338.
- C2.129 *SMULL* on page C2-339.
- C2.130 *SMULWy* on page C2-340.
- C2.131 *SMUSD* on page C2-341.
- C2.132 *SRS* on page C2-342.
- C2.133 *SSAT* on page C2-344.
- C2.134 *SSAT16* on page C2-345.
- C2.135 *SSAX* on page C2-346.

- [C2.136 SSUB8](#) on page C2-348.
- [C2.137 SSUB16](#) on page C2-350.
- [C2.138 STC and STC2](#) on page C2-352.
- [C2.139 STL](#) on page C2-354.
- [C2.140 STLEX](#) on page C2-355.
- [C2.141 STM](#) on page C2-357.
- [C2.142 STR \(immediate offset\)](#) on page C2-359.
- [C2.143 STR \(register offset\)](#) on page C2-361.
- [C2.144 STR, unprivileged](#) on page C2-363.
- [C2.145 STREX](#) on page C2-365.
- [C2.146 SUB](#) on page C2-367.
- [C2.147 SUBS pc, lr](#) on page C2-370.
- [C2.148 SVC](#) on page C2-372.
- [C2.149 SWP and SWPB](#) on page C2-373.
- [C2.150 SXTAB](#) on page C2-374.
- [C2.151 SXTAB16](#) on page C2-376.
- [C2.152 SXTAH](#) on page C2-378.
- [C2.153 SXTB](#) on page C2-380.
- [C2.154 SXTB16](#) on page C2-382.
- [C2.155 SXTH](#) on page C2-383.
- [C2.156 SYS](#) on page C2-385.
- [C2.157 TBB and TBH](#) on page C2-386.
- [C2.158 TEQ](#) on page C2-387.
- [C2.159 TST](#) on page C2-389.
- [C2.160 TT, TTT, TTA, TTAT](#) on page C2-391.
- [C2.161 UADD8](#) on page C2-393.
- [C2.162 UADD16](#) on page C2-395.
- [C2.163 UASX](#) on page C2-397.
- [C2.164 UBXF](#) on page C2-399.
- [C2.165 UDF](#) on page C2-400.
- [C2.166 UDIV](#) on page C2-401.
- [C2.167 UHADD8](#) on page C2-402.
- [C2.168 UHADD16](#) on page C2-403.
- [C2.169 UHASX](#) on page C2-404.
- [C2.170 UHSAX](#) on page C2-405.
- [C2.171 UHSUB8](#) on page C2-406.
- [C2.172 UHSUB16](#) on page C2-407.
- [C2.173 UMAAL](#) on page C2-408.
- [C2.174 UMLAL](#) on page C2-409.
- [C2.175 UMULL](#) on page C2-410.
- [C2.176 UQADD8](#) on page C2-411.
- [C2.177 UQADD16](#) on page C2-412.
- [C2.178 UQASX](#) on page C2-413.
- [C2.179 UQSAX](#) on page C2-414.
- [C2.180 UQSUB8](#) on page C2-415.
- [C2.181 UQSUB16](#) on page C2-416.
- [C2.182 USAD8](#) on page C2-417.
- [C2.183 USADA8](#) on page C2-418.
- [C2.184 USAT](#) on page C2-419.
- [C2.185 USAT16](#) on page C2-420.
- [C2.186 USAX](#) on page C2-421.
- [C2.187 USUB8](#) on page C2-423.
- [C2.188 USUB16](#) on page C2-425.
- [C2.189 UXTAB](#) on page C2-426.
- [C2.190 UXTAB16](#) on page C2-428.
- [C2.191 UXTAH](#) on page C2-430.

- *C2.192 UXTB* on page C2-432.
- *C2.193 UXTB16* on page C2-434.
- *C2.194 UXTH* on page C2-435.
- *C2.195 WFE* on page C2-437.
- *C2.196 WFI* on page C2-438.
- *C2.197 YIELD* on page C2-439.

## C2.1 A32 and T32 instruction summary

An overview of the instructions available in the A32 and T32 instruction sets.

**Table C2-1 Summary of instructions**

Mnemonic	Brief description
ADC, ADD	Add with Carry, Add
ADR	Load program or register-relative address (short range)
AND	Logical AND
ASR	Arithmetic Shift Right
B	Branch
BFC, BFI	Bit Field Clear and Insert
BIC	Bit Clear
BKPT	Software breakpoint
BL	Branch with Link
BLX, BLXNS	Branch with Link, change instruction set, Branch with Link and Exchange (Non-secure)
BX, BXNS	Branch, change instruction set, Branch and Exchange (Non-secure)
CBZ, CBNZ	Compare and Branch if {Non}Zero
CDP	Coprocessor Data Processing operation
CDP2	Coprocessor Data Processing operation
CLREX	Clear Exclusive
CLZ	Count leading zeros
CMN, CMP	Compare Negative, Compare
CPS	Change Processor State
CRC32	CRC32
CRC32C	CRC32C
CSDB	Consumption of Speculative Data Barrier
DBG	Debug
DCPS1	Debug switch to exception level 1
DCPS2	Debug switch to exception level 2
DCPS3	Debug switch to exception level 3
DMB, DSB	Data Memory Barrier, Data Synchronization Barrier
DSB	Data Synchronization Barrier
EOR	Exclusive OR
ERET	Exception Return
ESB	Error Synchronization Barrier
HLT	Halting breakpoint
HVC	Hypervisor Call

**Table C2-1 Summary of instructions (continued)**

Mnemonic	Brief description
ISB	Instruction Synchronization Barrier
IT	If-Then
LDAEX, LDAEXB, LDAEXH, LDAEXD	Load-Acquire Register Exclusive Word, Byte, Halfword, Doubleword
LDC, LDC2	Load Coprocessor
LDM	Load Multiple registers
LDR	Load Register with word
LDA, LDAB, LDAH	Load-Acquire Register Word, Byte, Halfword
LDRB	Load Register with Byte
LDRBT	Load Register with Byte, user mode
LDRD	Load Registers with two words
LDREX, LDREXB, LDREXH, LDREXD	Load Register Exclusive Word, Byte, Halfword, Doubleword
LDRH	Load Register with Halfword
LDRHT	Load Register with Halfword, user mode
LDRSB	Load Register with Signed Byte
LDRSBT	Load Register with Signed Byte, user mode
LDRSH	Load Register with Signed Halfword
LDRSHT	Load Register with Signed Halfword, user mode
LDRT	Load Register with word, user mode
LSL, LSR	Logical Shift Left, Logical Shift Right
MCR	Move from Register to Coprocessor
MCRR	Move from Registers to Coprocessor
MLA	Multiply Accumulate
MLS	Multiply and Subtract
MOV	Move
MOVT	Move Top
MRC	Move from Coprocessor to Register
MRRC	Move from Coprocessor to Registers
MRS	Move from PSR to Register
MSR	Move from Register to PSR
MUL	Multiply
MVN	Move Not
NOP	No Operation
ORN	Logical OR NOT
ORR	Logical OR
PKHBT, PKHTB	Pack Halfwords

**Table C2-1 Summary of instructions (continued)**

Mnemonic	Brief description
PLD	Preload Data
PLDW	Preload Data with intent to Write
PLI	Preload Instruction
PUSH, POP	PUSH registers to stack, POP registers from stack
QADD, QDADD, QDSUB, QSUB	Saturating arithmetic
QADD8, QADD16, QASX, QSUB8, QSUB16, QSAX	Parallel signed saturating arithmetic
RBIT	Reverse Bits
REV, REV16, REVSH	Reverse byte order
RFE	Return From Exception
ROR	Rotate Right Register
RRX	Rotate Right with Extend
RSB	Reverse Subtract
RSC	Reverse Subtract with Carry
SADD8, SADD16, SASX	Parallel Signed arithmetic
SBC	Subtract with Carry
SBFX, UBFX	Signed, Unsigned Bit Field eXtract
SDIV	Signed Divide
SEL	Select bytes according to APSR GE flags
SETEND	Set Endianness for memory accesses
SETPAN	Set Privileged Access Never
SEV	Set Event
SEVL	Set Event Locally
SG	Secure Gateway
SHADD8, SHADD16, SHASX, SHSUB8, SHSUB16, SHSAX	Parallel Signed Halving arithmetic
SMC	Secure Monitor Call
SMLAxy	Signed Multiply with Accumulate ( $32 \leq 16 \times 16 + 32$ )
SMLAD	Dual Signed Multiply Accumulate  ( $32 \leq 32 + 16 \times 16 + 16 \times 16$ )
SMLAL	Signed Multiply Accumulate ( $64 \leq 64 + 32 \times 32$ )
SMLALxy	Signed Multiply Accumulate ( $64 \leq 64 + 16 \times 16$ )
SMLALD	Dual Signed Multiply Accumulate Long  ( $64 \leq 64 + 16 \times 16 + 16 \times 16$ )
SMLAWy	Signed Multiply with Accumulate ( $32 \leq 32 \times 16 + 32$ )

**Table C2-1 Summary of instructions (continued)**

Mnemonic	Brief description
SMLSD	Dual Signed Multiply Subtract Accumulate $(32 \leq 32 + 16 \times 16 - 16 \times 16)$
SMLS LD	Dual Signed Multiply Subtract Accumulate Long $(64 \leq 64 + 16 \times 16 - 16 \times 16)$
SMMLA	Signed top word Multiply with Accumulate ( $32 \leq \text{TopWord}(32 \times 32 + 32)$ )
SMMLS	Signed top word Multiply with Subtract ( $32 \leq \text{TopWord}(32 - 32 \times 32)$ )
SMMUL	Signed top word Multiply ( $32 \leq \text{TopWord}(32 \times 32)$ )
SMUAD, SMUSD	Dual Signed Multiply, and Add or Subtract products
SMULxy	Signed Multiply ( $32 \leq 16 \times 16$ )
SMULL	Signed Multiply ( $64 \leq 32 \times 32$ )
SMULW y	Signed Multiply ( $32 \leq 32 \times 16$ )
SRS	Store Return State
SSAT	Signed Saturate
SSAT16	Signed Saturate, parallel halfwords
SSUB8, SSUB16, SSAX	Parallel Signed arithmetic
STC	Store Coprocessor
STM	Store Multiple registers
STR	Store Register with word
STRB	Store Register with Byte
STRBT	Store Register with Byte, user mode
STRD	Store Registers with two words
STREX, STREXB, STREXH, STREXD	Store Register Exclusive Word, Byte, Halfword, Doubleword
STRH	Store Register with Halfword
STRHT	Store Register with Halfword, user mode
STL, STL B, STL H	Store-Release Word, Byte, Halfword
STLEX, STLEXB, STLEXH, STLEXD	Store-Release Exclusive Word, Byte, Halfword, Doubleword
STRT	Store Register with word, user mode
SUB	Subtract
SUBS pc, lr	Exception return, no stack
SVC (formerly SWI)	Supervisor Call
SXTAB, SXTAB16, SXTAH	Signed extend, with Addition
SXTB, SXTH	Signed extend
SXTB16	Signed extend
SYS	Execute System coprocessor instruction
TBB, TBH	Table Branch Byte, Halfword

**Table C2-1 Summary of instructions (continued)**

Mnemonic	Brief description
TEQ	Test Equivalence
TST	Test
TT, TTT, TTA, TTAT	Test Target (Alternate Domain, Unprivileged)
UADD8, UADD16, UASX	Parallel Unsigned arithmetic
UDF	Permanently Undefined
UDIV	Unsigned Divide
UHADD8, UHADD16, UHASX, UHSUB8, UHSUB16, UHSAX	Parallel Unsigned Halving arithmetic
UMAAL	Unsigned Multiply Accumulate Accumulate Long  ( $64 \leq 32 + 32 + 32 \times 32$ )
UMLAL, UMULL	Unsigned Multiply Accumulate, Unsigned Multiply  ( $64 \leq 32 \times 32 + 64$ ), ( $64 \leq 32 \times 32$ )
UQADD8, UQADD16, UQASX, UQSUB8, UQSUB16, UQSAX	Parallel Unsigned Saturating arithmetic
USAD8	Unsigned Sum of Absolute Differences
USADA8	Accumulate Unsigned Sum of Absolute Differences
USAT	Unsigned Saturate
USAT16	Unsigned Saturate, parallel halfwords
USUB8, USUB16, USAX	Parallel Unsigned arithmetic
UXTAB, UXTAB16, UXTAH	Unsigned extend with Addition
UXTB, UXTH	Unsigned extend
UXTB16	Unsigned extend
V*	See <a href="#">Chapter C3 Advanced SIMD Instructions (32-bit)</a> on page <a href="#">C3-441</a> and <a href="#">Chapter C4 Floating-point Instructions (32-bit)</a> on page <a href="#">C4-599</a>
WFE, WFI, YIELD	Wait For Event, Wait For Interrupt, Yield

## C2.2 Instruction width specifiers

The instruction width specifiers .W and .N control the size of T32 instruction encodings.

In T32 code the .W width specifier forces the assembler to generate a 32-bit encoding, even if a 16-bit encoding is available. The .W specifier has no effect when assembling to A32 code.

In T32 code the .N width specifier forces the assembler to generate a 16-bit encoding. In this case, if the instruction cannot be encoded in 16 bits or if .N is used in A32 code, the assembler generates an error.

If you use an instruction width specifier, you must place it immediately after the instruction mnemonic and any condition code, for example:

```
BCS.W    label    ; forces 32-bit instruction even for a short branch  
B.N     label    ; faults if label out of range for 16-bit instruction
```

## C2.3 Flexible second operand (Operand2)

Many A32 and T32 general data processing instructions have a flexible second operand.

This is shown as *Operand2* in the descriptions of the syntax of each instruction.

*Operand2* can be a:

- Constant.
- Register with optional shift.

### ***Related concepts***

[C2.6 Shift operations on page C2-165](#)

### ***Related reference***

[C2.4 Syntax of Operand2 as a constant on page C2-163](#)

[C2.5 Syntax of Operand2 as a register with optional shift on page C2-164](#)

## C2.4 Syntax of Operand2 as a constant

An Operand2 constant in an instruction has a limited range of values.

### Syntax

`#constant`

where *constant* is an expression evaluating to a numeric value.

### Usage

In A32 instructions, *constant* can have any value that can be produced by rotating an 8-bit value right by any even number of bits within a 32-bit word.

In T32 instructions, *constant* can be:

- Any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word.
- Any constant of the form `0x00XY00XY`.
- Any constant of the form `0xXY00XY00`.
- Any constant of the form `0xXYXYXYXY`.

---

#### Note

---

In these constants, X and Y are hexadecimal digits.

---

In addition, in a small number of instructions, *constant* can take a wider range of values. These are listed in the individual instruction descriptions.

When an Operand2 constant is used with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to bit[31] of the constant, if the constant is greater than 255 and can be produced by shifting an 8-bit value. These instructions do not affect the carry flag if Operand2 is any other constant.

### Instruction substitution

If the value of an Operand2 constant is not available, but its logical inverse or negation is available, then the assembler produces an equivalent instruction and inverts or negates the constant.

For example, an assembler might assemble the instruction `CMP Rd, #0xFFFFFFFF` as the equivalent instruction `CMN Rd, #0x2`.

Be aware of this when comparing disassembly listings with source code.

### Related concepts

[C2.6 Shift operations](#) on page C2-165

### Related reference

[C2.3 Flexible second operand \(Operand2\)](#) on page C2-162

[C2.5 Syntax of Operand2 as a register with optional shift](#) on page C2-164

## C2.5 Syntax of Operand2 as a register with optional shift

When you use an Operand2 register in an instruction, you can optionally also specify a shift value.

### Syntax

*Rm {, shift}*

where:

*Rm*

is the register holding the data for the second operand.

*shift*

is an optional constant or register-controlled shift to be applied to *Rm*. It can be one of:

*ASR #n*

arithmetic shift right *n* bits,  $1 \leq n \leq 32$ .

*LSL #n*

logical shift left *n* bits,  $1 \leq n \leq 31$ .

*LSR #n*

logical shift right *n* bits,  $1 \leq n \leq 32$ .

*ROR #n*

rotate right *n* bits,  $1 \leq n \leq 31$ .

**RRX**

rotate right one bit, with extend.

*type Rs*

register-controlled shift is available in Arm code only, where:

*type*

is one of ASR, LSL, LSR, ROR.

*Rs*

is a register supplying the shift amount, and only the least significant byte is used.

- if omitted, no shift occurs, equivalent to LSL #0.

### Usage

If you omit the shift, or specify LSL #0, the instruction uses the value in *Rm*.

If you specify a shift, the shift is applied to the value in *Rm*, and the resulting 32-bit value is used by the instruction. However, the contents of the register *Rm* remain unchanged. Specifying a register with shift also updates the carry flag when used with certain instructions.

### Related concepts

[C2.6 Shift operations on page C2-165](#)

### Related reference

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.4 Syntax of Operand2 as a constant on page C2-163](#)

## C2.6 Shift operations

Register shift operations move the bits in a register left or right by a specified number of bits, called the shift length.

Register shift can be performed:

- Directly by the instructions ASR, LSR, LSL, ROR, and RRX, and the result is written to a destination register.
- During the calculation of *Operand2* by the instructions that specify the second operand as a register with shift. The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction, see the individual instruction description or the flexible second operand description. If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0.

### Arithmetic shift right (ASR)

Arithmetic shift right by  $n$  bits moves the left-hand 32- $n$  bits of a register to the right by  $n$  places, into the right-hand 32- $n$  bits of the result. It copies the original bit[31] of the register into the left-hand  $n$  bits of the result.

You can use the ASR  $\#n$  operation to divide the value in the register  $Rm$  by  $2^n$ , with the result being rounded towards negative-infinity.

When the instruction is ASRS or when ASR  $\#n$  is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[ $n-1$ ], of the register  $Rm$ .

————— Note —————

- If  $n$  is 32 or more, then all the bits in the result are set to the value of bit[31] of  $Rm$ .
- If  $n$  is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of  $Rm$ .

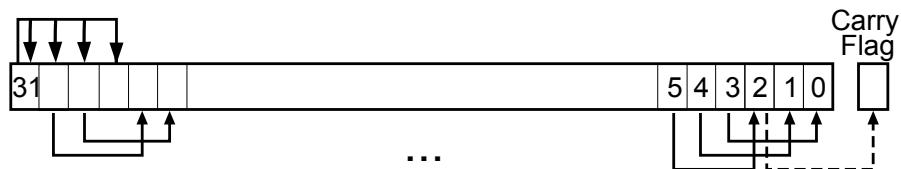


Figure C2-1 ASR #3

### Logical shift right (LSR)

Logical shift right by  $n$  bits moves the left-hand 32- $n$  bits of a register to the right by  $n$  places, into the right-hand 32- $n$  bits of the result. It sets the left-hand  $n$  bits of the result to 0.

You can use the LSR  $\#n$  operation to divide the value in the register  $Rm$  by  $2^n$ , if the value is regarded as an unsigned integer.

When the instruction is LSRS or when LSR  $\#n$  is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[ $n-1$ ], of the register  $Rm$ .

————— Note —————

- If  $n$  is 32 or more, then all the bits in the result are cleared to 0.
- If  $n$  is 33 or more and the carry flag is updated, it is updated to 0.

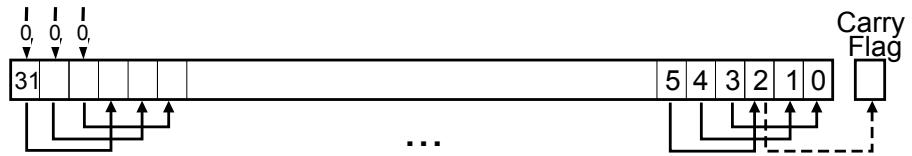


Figure C2-2 LSR #3

### Logical shift left (LSL)

Logical shift left by  $n$  bits moves the right-hand 32- $n$  bits of a register to the left by  $n$  places, into the left-hand 32- $n$  bits of the result. It sets the right-hand  $n$  bits of the result to 0.

You can use the LSL  $\#n$  operation to multiply the value in the register  $Rm$  by  $2^n$ , if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is LSLS or when LSL  $\#n$ , with non-zero  $n$ , is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[32- $n$ ], of the register  $Rm$ . These instructions do not affect the carry flag when used with LSL  $\#0$ .

————— Note —————

- If  $n$  is 32 or more, then all the bits in the result are cleared to 0.
- If  $n$  is 33 or more and the carry flag is updated, it is updated to 0.

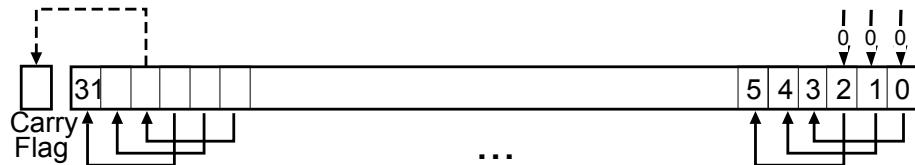


Figure C2-3 LSL #3

### Rotate right (ROR)

Rotate right by  $n$  bits moves the left-hand 32- $n$  bits of a register to the right by  $n$  places, into the right-hand 32- $n$  bits of the result. It also moves the right-hand  $n$  bits of the register into the left-hand  $n$  bits of the result.

When the instruction is RORS or when ROR  $\#n$  is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit rotation, bit[n-1], of the register  $Rm$ .

————— Note —————

- If  $n$  is 32, then the value of the result is same as the value in  $Rm$ , and if the carry flag is updated, it is updated to bit[31] of  $Rm$ .
- ROR with shift length,  $n$ , more than 32 is the same as ROR with shift length  $n-32$ .

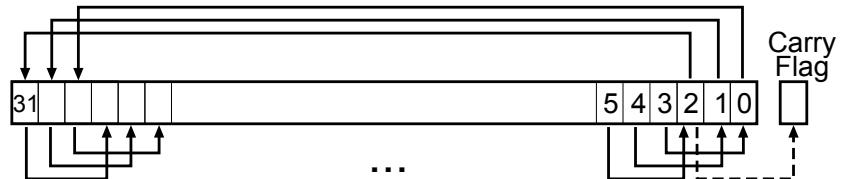


Figure C2-4 ROR #3

### Rotate right with extend (RRX)

Rotate right with extend moves the bits of a register to the right by one bit. It copies the carry flag into bit[31] of the result.

When the instruction is RRXS or when RRX is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to bit[0] of the register *Rm*.

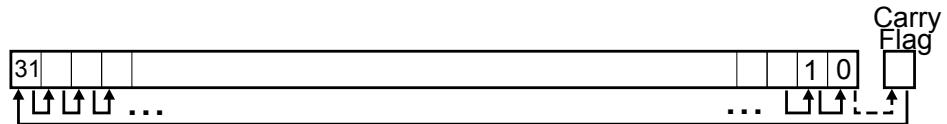


Figure C2-5 RRX

#### Related reference

[C2.3 Flexible second operand \(\*Operand2\*\) on page C2-162](#)

[C2.4 Syntax of \*Operand2\* as a constant on page C2-163](#)

[C2.5 Syntax of \*Operand2\* as a register with optional shift on page C2-164](#)

## C2.7 Saturating instructions

Some A32 and T32 instructions perform saturating arithmetic.

The saturating instructions are:

- QADD.
- QDADD.
- QDSUB.
- QSUB.
- SSAT.
- USAT.

Some of the parallel instructions are also saturating.

### Saturating arithmetic

Saturation means that, for some value of  $2^n$  that depends on the instruction:

- For a signed saturating operation, if the full result would be less than  $-2^n$ , the result returned is  $-2^n$ .
- For an unsigned saturating operation, if the full result would be negative, the result returned is zero.
- If the full result would be greater than  $2^n-1$ , the result returned is  $2^n-1$ .

When any of these occurs, it is called saturation. Some instructions set the Q flag when saturation occurs.

---

————— Note ————

Saturating instructions do not clear the Q flag when saturation does not occur. To clear the Q flag, use an MSR instruction.

---

The Q flag can also be set by two other instructions, but these instructions do not saturate.

#### *Related reference*

[C2.78 QADD on page C2-276](#)

[C2.85 QSUB on page C2-283](#)

[C2.82 QDADD on page C2-280](#)

[C2.83 QDSUB on page C2-281](#)

[C2.116 SMLAxy on page C2-324](#)

[C2.121 SMLAWy on page C2-331](#)

[C2.128 SMULxy on page C2-338](#)

[C2.130 SMULWy on page C2-340](#)

[C2.133 SSAT on page C2-344](#)

[C2.184 USAT on page C2-419](#)

[C2.68 MSR \(general-purpose register to PSR\) on page C2-261](#)

## C2.8 ADC

Add with Carry.

### Syntax

ADC{S}{cond} {Rd}, Rn, Operand2

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Operand2*

is a flexible second operand.

### Usage

The ADC (Add with Carry) instruction adds the values in *Rn* and *Operand2*, together with the carry flag.

You can use ADC to synthesize multiword arithmetic.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

### Use of PC and SP in T32 instructions

You cannot use PC (R15) for *Rd*, or any operand with the ADC command.

You cannot use SP (R13) for *Rd*, or any operand with the ADC command.

### Use of PC and SP in A32 instructions

You cannot use PC for *Rd* or any operand in any data processing instruction that has a register-controlled shift.

Use of PC for any operand, in instructions without register-controlled shift, is deprecated.

If you use PC (R15) as *Rn* or *Operand2*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc,lr instruction.

Use of SP with the ADC A32 instruction is deprecated.

### Condition flags

If S is specified, the ADC instruction updates the N, Z, C and V flags according to the result.

### 16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

**ADCS Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used outside an IT block.

**ADC{cond} Rd, Rd, Rm**

Rd and Rm must both be Lo registers. This form can only be used inside an IT block.

**Multiword arithmetic examples**

These two instructions add a 64-bit integer contained in R2 and R3 to another 64-bit integer contained in R0 and R1, and place the result in R4 and R5.

```
ADDS    r4, r0, r2    ; adding the least significant words
ADC     r5, r1, r3    ; adding the most significant words
```

**Related reference**

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.9 ADD

Add without Carry.

### Syntax

`ADD{S}{cond} {Rd}, Rn, Operand2`  
`ADD{cond} {Rd}, Rn, #imm12 ; T32, 32-bit encoding only`  
where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

**cond**

is an optional condition code.

**Rd**

is the destination register.

**Rn**

is the register holding the first operand.

**Operand2**

is a flexible second operand.

**imm12**

is any value in the range 0-4095.

### Operation

The ADD instruction adds the values in *Rn* and *Operand2* or *imm12*.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

### Use of PC and SP in T32 instructions

Generally, you cannot use PC (R15) for *Rd*, or any operand.

The exceptions are:

- you can use PC for *Rn* in 32-bit encodings of T32 ADD instructions, with a constant *Operand2* value in the range 0-4095, and no S suffix. These instructions are useful for generating PC-relative addresses. Bit[1] of the PC value reads as 0 in this case, so that the base address for the calculation is always word-aligned.
- you can use PC in 16-bit encodings of T32 ADD{cond} Rd, Rd, Rm instructions, where both registers cannot be PC. However, the following 16-bit T32 instructions are deprecated:
  - ADD{cond} PC, SP, PC.
  - ADD{cond} SP, SP, PC.

Generally, you cannot use SP (R13) for *Rd*, or any operand. Except that:

- You can use SP for *Rn* in ADD instructions.
- ADD{cond} SP, SP, SP is permitted but is deprecated in Armv6T2 and above.
- ADD{S}{cond} SP, SP, Rm{,shift} and SUB{S}{cond} SP, SP, Rm{,shift} are permitted if *shift* is omitted or LSL #1, LSL #2, or LSL #3.

### Use of PC and SP in A32 instructions

You cannot use PC for *Rd* or any operand in any data processing instruction that has a register-controlled shift.

In ADD instructions without register-controlled shift, use of PC is deprecated except for the following cases:

- Use of PC for *Rd* in instructions that do not add SP to a register.
- Use of PC for *Rn* and use of PC for *Rm* in instructions that add two registers other than SP.
- Use of PC for *Rn* in the instruction **ADD{cond} Rd, Rn, #Constant**.

If you use PC (*R15*) as *Rn* or *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the **SUBS pc, lr** instruction.

You can use SP for *Rn* in ADD instructions, however, **ADDS PC, SP, #Constant** is deprecated.

You can use SP in ADD (register) if *Rn* is SP and *shift* is omitted or LSL #1, LSL #2, or LSL #3.

Other uses of SP in these A32 instructions are deprecated.

### Condition flags

If S is specified, these instructions update the N, Z, C and V flags according to the result.

### 16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

**ADDS Rd, Rn, #imm**

*imm* range 0-7. *Rd* and *Rn* must both be Lo registers. This form can only be used outside an IT block.

**ADD{cond} Rd, Rn, #imm**

*imm* range 0-7. *Rd* and *Rn* must both be Lo registers. This form can only be used inside an IT block.

**ADDS Rd, Rn, Rm**

*Rd*, *Rn* and *Rm* must all be Lo registers. This form can only be used outside an IT block.

**ADD{cond} Rd, Rn, Rm**

*Rd*, *Rn* and *Rm* must all be Lo registers. This form can only be used inside an IT block.

**ADDS Rd, Rd, #imm**

*imm* range 0-255. *Rd* must be a Lo register. This form can only be used outside an IT block.

**ADD{cond} Rd, Rd, #imm**

*imm* range 0-255. *Rd* must be a Lo register. This form can only be used inside an IT block.

**ADD SP, SP, #imm**

*imm* range 0-508, word aligned.

**ADD Rd, SP, #imm**

*imm* range 0-1020, word aligned. *Rd* must be a Lo register.

**ADD Rd, pc, #imm**

*imm* range 0-1020, word aligned. *Rd* must be a Lo register. Bits[1:0] of the PC are read as 0 in this instruction.

### Example

```
ADD      r2, r1, r3
```

### Multiword arithmetic example

These two instructions add a 64-bit integer contained in R2 and R3 to another 64-bit integer contained in R0 and R1, and place the result in R4 and R5.

ADDS	r4, r0, r2	; adding the least significant words
ADC	r5, r1, r3	; adding the most significant words

**Related reference**

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C1.9 Condition code suffixes on page C1-142](#)

[C2.147 SUBS pc, lr on page C2-370](#)

## C2.10 ADR (PC-relative)

Generate a PC-relative address in the destination register, for a label in the current area.

### Syntax

`ADR{cond}{.W} Rd,Label`

where:

**cond**

is an optional condition code.

**.W**

is an optional instruction width specifier.

**Rd**

is the destination register to load.

**Label**

is a PC-relative expression.

*Label* must be within a limited distance of the current instruction.

### Usage

ADR produces position-independent code, because the assembler generates an instruction that adds or subtracts a value to the PC.

*Label* must evaluate to an address in the same assembler area as the ADR instruction.

If you use ADR to generate a target for a BX or BLX instruction, it is your responsibility to set the T32 bit (bit 0) of the address if the target contains T32 instructions.

### Offset range and architectures

The assembler calculates the offset from the PC for you. The assembler generates an error if *Label* is out of range.

The following table shows the possible offsets between the label and the current instruction:

**Table C2-2 PC-relative offsets**

Instruction	Offset range
A32 ADR	See <a href="#">C2.4 Syntax of Operand2 as a constant</a> on page C2-163.
T32 ADR, 32-bit encoding	$\pm 4095$
T32 ADR, 16-bit encoding <sup>a</sup>	0-1020 <sup>b</sup>

### ADR in T32

You can use the .W width specifier to force ADR to generate a 32-bit instruction in T32 code. ADR with .W always generates a 32-bit instruction, even if the address can be generated in a 16-bit instruction.

For forward references, ADR without .W always generates a 16-bit instruction in T32 code, even if that results in failure for an address that could be generated in a 32-bit T32 ADD instruction.

### Restrictions

In T32 code, *Rd* cannot be PC or SP.

In A32 code, *Rd* can be PC or SP but use of SP is deprecated.

<sup>a</sup> Rd must be in the range R0-R7.

<sup>b</sup> Must be a multiple of 4.

**Related reference**

[C2.4 Syntax of Operand2 as a constant on page C2-163](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.11 ADR (register-relative)

Generate a register-relative address in the destination register, for a label defined in a storage map.

### Syntax

`ADR{cond}{ .W} Rd,Label`

where:

*cond*

is an optional condition code.

*.W*

is an optional instruction width specifier.

*Rd*

is the destination register to load.

*Label*

is a symbol defined by the FIELD directive. *Label* specifies an offset from the base register which is defined using the MAP directive.

*Label* must be within a limited distance from the base register.

### Usage

ADR generates code to easily access named fields inside a storage map.

### Restrictions

In T32 code:

- *Rd* cannot be PC.
- *Rd* can be SP only if the base register is SP.

### Offset range and architectures

The assembler calculates the offset from the base register for you. The assembler generates an error if *Label* is out of range.

The following table shows the possible offsets between the label and the current instruction:

Table C2-3 Register-relative offsets

Instruction	Offset range
A32 ADR	See <a href="#">C2.4 Syntax of Operand2 as a constant on page C2-163</a>
T32 ADR, 32-bit encoding	$\pm 4095$
T32 ADR, 16-bit encoding, base register is SP <sup>c</sup>	0-1020 <sup>d</sup>

### ADR in T32

You can use the *.W* width specifier to force ADR to generate a 32-bit instruction in T32 code. ADR with *.W* always generates a 32-bit instruction, even if the address can be generated in a 16-bit instruction.

For forward references, ADR without *.W*, with base register SP, always generates a 16-bit instruction in T32 code, even if that results in failure for an address that could be generated in a 32-bit T32 ADD instruction.

### Related reference

[C2.4 Syntax of Operand2 as a constant on page C2-163](#)

<sup>c</sup> *Rd* must be in the range R0-R7 or SP. If *Rd* is SP, the offset range is -508 to 508 and must be a multiple of 4

<sup>d</sup> Must be a multiple of 4.

*C1.9 Condition code suffixes* on page C1-142

## C2.12 AND

Logical AND.

### Syntax

`AND{S}{cond} Rd, Rn, Operand2`

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

**cond**

is an optional condition code.

**Rd**

is the destination register.

**Rn**

is the register holding the first operand.

**Operand2**

is a flexible second operand.

### Operation

The AND instruction performs bitwise AND operations on the values in *Rn* and *Operand2*.

In certain circumstances, the assembler can substitute BIC for AND, or AND for BIC. Be aware of this when reading disassembly listings.

### Use of PC in T32 instructions

You cannot use PC (R15) for *Rd* or any operand with the AND instruction.

### Use of PC and SP in A32 instructions

You can use PC and SP with the AND A32 instruction but this is deprecated.

If you use PC as *Rn*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc, lr instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

### Condition flags

If S is specified, the AND instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

### 16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

**ANDS Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used outside an IT block.

**AND{cond} Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used inside an IT block.

It does not matter if you specify AND{S} *Rd, Rm, Rd*. The instruction is the same.

## Examples

```
AND      r9,r2,#0xFF00
ANDS    r9, r8, #0x19
```

### Related reference

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.147 SUBS pc, lr on page C2-370](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.13 ASR

Arithmetic Shift Right. This instruction is a preferred synonym for MOV instructions with shifted register operands.

### Syntax

```
ASR{S}{cond} Rd, Rm, Rs  
ASR{S}{cond} Rd, Rm, #sh
```

where:

*S*

is an optional suffix. If *S* is specified, the condition flags are updated on the result of the operation.

*Rd*

is the destination register.

*Rm*

is the register holding the first operand. This operand is shifted right.

*Rs*

is a register holding a shift value to apply to the value in *Rm*. Only the least significant byte is used.

*sh*

is a constant shift. The range of values permitted is 1-32.

### Operation

ASR provides the signed value of the contents of a register divided by a power of two. It copies the sign bit into vacated bit positions on the left.

### Restrictions in T32 code

T32 instructions must not use PC or SP.

### Use of SP and PC in A32 instructions

You can use SP in the ASR A32 instruction but this is deprecated.

You cannot use PC in instructions with the ASR{S}{cond} Rd, Rm, Rs syntax. You can use PC for *Rd* and *Rm* in the other syntax, but this is deprecated.

If you use PC as *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the *S* suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

#### Note

The A32 instruction ASRS{cond} pc,Rm,#sh always disassembles to the preferred form MOVS{cond} pc,Rm{,shift}.

#### Caution

Do not use the *S* suffix when using PC as *Rd* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *Rd* or any operand in the ASR instruction if it has a register-controlled shift.

## Condition flags

If S is specified, the ASR instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

## 16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

**ASRS Rd, Rm, #sh**

Rd and Rm must both be Lo registers. This form can only be used outside an IT block.

**ASR{cond} Rd, Rm, #sh**

Rd and Rm must both be Lo registers. This form can only be used inside an IT block.

**ASRS Rd, Rd, Rs**

Rd and Rs must both be Lo registers. This form can only be used outside an IT block.

**ASR{cond} Rd, Rd, Rs**

Rd and Rs must both be Lo registers. This form can only be used inside an IT block.

## Architectures

This instruction is available in A32 and T32.

## Example

```
ASR      r7, r8, r9
```

### Related reference

[C2.61 MOV on page C2-252](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.14 B

Branch.

### Syntax

`B{cond}{.W} Label`

where:

*cond*

is an optional condition code.

*.W*

is an optional instruction width specifier to force the use of a 32-bit B instruction in T32.

*Label*

is a PC-relative expression.

### Operation

The B instruction causes a branch to *Label*.

### Instruction availability and branch ranges

The following table shows the branch ranges that are available in A32 and T32 code. Instructions that are not shown in this table are not available.

Table C2-4 B instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
B label	$\pm 32\text{MB}$	$\pm 2\text{KB}$	$\pm 16\text{MB}$ <sup>e</sup>
B{cond} label	$\pm 32\text{MB}$	-252 to +258	$\pm 1\text{MB}$ <sup>e</sup>

### Extending branch ranges

Machine-level B instructions have restricted ranges from the address of the current instruction. However, you can use these instructions even if *Label* is out of range. Often you do not know where the linker places *Label*. When necessary, the linker adds code to enable longer branches. The added code is called a veneer.

### B in T32

You can use the .W width specifier to force B to generate a 32-bit instruction in T32 code.

B.W always generates a 32-bit instruction, even if the target could be reached using a 16-bit instruction.

For forward references, B without .W always generates a 16-bit instruction in T32 code, even if that results in failure for a target that could be reached using a 32-bit T32 instruction.

### Condition flags

The B instruction does not change the flags.

### Architectures

See the earlier table for details of availability of the B instruction.

### Example

B	loopA
---	-------

<sup>e</sup> Use .W to instruct the assembler to use this 32-bit instruction.

***Related reference***

*C1.9 Condition code suffixes on page C1-142*

## C2.15 BFC

Bit Field Clear.

### Syntax

BFC{*cond*} *Rd*, #*Lsb*, #*width*

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Lsb*

is the least significant bit that is to be cleared.

*width*

is the number of bits to be cleared. *width* must not be 0, and (*width*+*Lsb*) must be less than or equal to 32.

### Operation

Clears adjacent bits in a register. *width* bits in *Rd* are cleared, starting at *Lsb*. Other bits in *Rd* are unchanged.

### Register restrictions

You cannot use PC for any register.

You can use SP in the BFC A32 instruction but this is deprecated. You cannot use SP in the BFC T32 instruction.

### Condition flags

The BFC instruction does not change the flags.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.16 BFI

Bit Field Insert.

### Syntax

`BFI{cond} Rd, Rn, #Lsb, #width`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the source register.

*Lsb*

is the least significant bit that is to be copied.

*width*

is the number of bits to be copied. *width* must not be 0, and (*width+Lsb*) must be less than or equal to 32.

### Operation

Inserts adjacent bits from one register into another. *width* bits in *Rd*, starting at *Lsb*, are replaced by *width* bits from *Rn*, starting at bit[0]. Other bits in *Rd* are unchanged.

### Register restrictions

You cannot use PC for any register.

You can use SP in the BFI A32 instruction but this is deprecated. You cannot use SP in the BFI T32 instruction.

### Condition flags

The BFI instruction does not change the flags.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.17 BIC

Bit Clear.

### Syntax

**BIC{S}{cond} Rd, Rn, Operand2**

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

**cond**

is an optional condition code.

**Rd**

is the destination register.

**Rn**

is the register holding the first operand.

**Operand2**

is a flexible second operand.

### Operation

The BIC (Bit Clear) instruction performs an AND operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*.

In certain circumstances, the assembler can substitute BIC for AND, or AND for BIC. Be aware of this when reading disassembly listings.

### Use of PC in T32 instructions

You cannot use PC (R15) for *Rd* or any operand in a BIC instruction.

### Use of PC and SP in A32 instructions

You can use PC and SP with the BIC instruction but they are deprecated.

If you use PC as *Rn*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc,lr instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

### Condition flags

If S is specified, the BIC instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

### 16-bit instructions

The following forms of the BIC instruction are available in T32 code, and are 16-bit instructions:

**BICS Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used outside an IT block.

**BIC{cond} Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used inside an IT block.

## Example

```
BIC      r0, r1, #0xab
```

### Related reference

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.147 SUBS pc, lr on page C2-370](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.18 BKPT

Breakpoint.

### Syntax

`BKPT #imm`

where:

*imm*

is an expression evaluating to an integer in the range:

- 0-65535 (a 16-bit value) in an A32 instruction.
- 0-255 (an 8-bit value) in a 16-bit T32 instruction.

### Usage

The BKPT instruction causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

In both A32 state and T32 state, *imm* is ignored by the Arm hardware. However, a debugger can use it to store additional information about the breakpoint.

BKPT is an unconditional instruction. It must not have a condition code in A32 code. In T32 code, the BKPT instruction does not require a condition code suffix because BKPT always executes irrespective of its condition code suffix.

### Architectures

This instruction is available in A32 and T32.

In T32, it is only available as a 16-bit instruction.

## C2.19 BL

Branch with Link.

### Syntax

`BL{cond}{.W} Label`

where:

*cond*

is an optional condition code. *cond* is not available on all forms of this instruction.

.W

is an optional instruction width specifier to force the use of a 32-bit BL instruction in T32.

*Label*

is a PC-relative expression.

### Operation

The BL instruction causes a branch to *Label*, and copies the address of the next instruction into LR (R14, the link register).

### Instruction availability and branch ranges

The following table shows the BL instructions that are available in A32 and T32 state. Instructions that are not shown in this table are not available.

Table C2-5 BL instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
BL label	±32MB	±4MB <sup>f</sup>	±16MB
BL{cond} label	±32MB	-	-

### Extending branch ranges

Machine-level BL instructions have restricted ranges from the address of the current instruction.

However, you can use these instructions even if *Label* is out of range. Often you do not know where the linker places *Label*. When necessary, the linker adds code to enable longer branches. The added code is called a veneer.

### Condition flags

The BL instruction does not change the flags.

### Availability

See the preceding table for details of availability of the BL instruction in both instruction sets.

### Examples

```
BLE      ng+8
BL       subC
BLLT    rtx
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

<sup>f</sup> BL label and BLX label are an instruction pair.

## C2.20 BLX, BLXNS

Branch with Link and exchange instruction set and Branch with Link and Exchange (Non-secure).

### Syntax

```
BLX{cond}{q} Label
BLX{cond}{q} Rm
BLXNS{cond}{q} Rm (Armv8-M only)
```

Where:

**cond**

Is an optional condition code. *cond* is not available on all forms of this instruction.

**q**

Is an optional instruction width specifier. Must be set to .W when *Label* is used.

**Label**

Is a PC-relative expression.

**Rm**

Is a register containing an address to branch to.

### Operation

The BLX instruction causes a branch to *Label*, or to the address contained in *Rm*. In addition:

- The BLX instruction copies the address of the next instruction into LR (R14, the link register).
- The BLX instruction can change the instruction set.

BLX *Label* always changes the instruction set. It changes a processor in A32 state to T32 state, or a processor in T32 state to A32 state.

BLX *Rm* derives the target instruction set from bit[0] of *Rm*:

- If bit[0] of *Rm* is 0, the processor changes to, or remains in, A32 state.
- If bit[0] of *Rm* is 1, the processor changes to, or remains in, T32 state.

———— Note ————

- There are no equivalent instructions to BLX to change between AArch32 and AArch64 state. The only way to change execution state is by a change of exception level.
- Armv8-M, Armv7-M, and Armv6-M only support the T32 instruction set. An attempt to change the instruction execution state causes the processor to take an exception on the instruction at the target address.

The BLXNS instruction calls a subroutine at an address and instruction set specified by a register, and causes a transition from the Secure to the Non-secure domain. This variant of the instruction must only be used when additional steps required to make such a transition safe are taken.

### Instruction availability and branch ranges

The following table shows the instructions that are available in A32 and T32 state. Instructions that are not shown in this table are not available.

Table C2-6 BLX instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
BLX <i>label</i>	±32MB	±4MB <sup>g</sup>	±16MB
BLX <i>Rm</i>	Available	Available	Use 16-bit

<sup>g</sup> BLX *label* and BL *label* are an instruction pair.

Table C2-6 BLX instruction availability and range (continued)

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
BLX{cond} Rm	Available	-	-
BLXNS	-	Available	-

### Register restrictions

You can use PC for *Rm* in the A32 BLX instruction, but this is deprecated. You cannot use PC in other A32 instructions.

You can use PC for *Rm* in the T32 BLX instruction. You cannot use PC in other T32 instructions.

You can use SP for *Rm* in this A32 instruction but this is deprecated.

You can use SP for *Rm* in the T32 BLX and BLXNS instructions, but this is deprecated. You cannot use SP in the other T32 instructions.

### Condition flags

These instructions do not change the flags.

### Availability

See the preceding table for details of availability of the BLX and BLXNS instructions in both instruction sets.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

[C2.2 Instruction width specifiers on page C2-161](#)

## C2.21 BX, BXNS

Branch and exchange instruction set and Branch and Exchange Non-secure.

### Syntax

`BX{cond}{q} Rm`

`BXNS{cond}{q} Rm` (Armv8-M only)

Where:

*cond*

Is an optional condition code. *cond* is not available on all forms of this instruction.

*q*

Is an optional instruction width specifier.

*Rm*

Is a register containing an address to branch to.

### Operation

The `BX` instruction causes a branch to the address contained in *Rm* and exchanges the instruction set, if necessary. The `BX` instruction can change the instruction set.

`BX Rm` derives the target instruction set from bit[0] of *Rm*:

- If bit[0] of *Rm* is 0, the processor changes to, or remains in, A32 state.
- If bit[0] of *Rm* is 1, the processor changes to, or remains in, T32 state.

---

#### Note

---

- There are no equivalent instructions to `BX` to change between AArch32 and AArch64 state. The only way to change execution state is by a change of exception level.
  - Armv8-M, Armv7-M, and Armv6-M only support the T32 instruction set. An attempt to change the instruction execution state causes the processor to take an exception on the instruction at the target address.
- 

`BX` can also be used for an exception return.

The `BXNS` instruction causes a branch to an address and instruction set specified by a register, and causes a transition from the Secure to the Non-secure domain. This variant of the instruction must only be used when additional steps required to make such a transition safe are taken.

### Instruction availability and branch ranges

The following table shows the instructions that are available in A32 and T32 state. Instructions that are not shown in this table are not available.

Table C2-7 BX instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
<code>BX Rm</code>	Available	Available	Use 16-bit
<code>BX{cond} Rm</code>	Available	-	-
<code>BXNS</code>	-	Available	-

### Register restrictions

You can use PC for *Rm* in the A32 `BX` instruction, but this is deprecated. You cannot use PC in other A32 instructions.

You can use PC for  $Rm$  in the T32 BX and BXNS instructions. You cannot use PC in other T32 instructions.

You can use SP for  $Rm$  in the A32 BX instruction but this is deprecated.

You can use SP for  $Rm$  in the T32 BX and BXNS instructions, but this is deprecated.

### Condition flags

These instructions do not change the flags.

### Availability

See the preceding table for details of availability of the BX and BXNS instructions in both instruction sets.

#### *Related reference*

[C1.9 Condition code suffixes on page C1-142](#)

[C2.2 Instruction width specifiers on page C2-161](#)

## C2.22 BXJ

Branch and change to Jazelle state.

### Syntax

`BXJ{cond} Rm`

where:

*cond*

is an optional condition code. *cond* is not available on all forms of this instruction.

*Rm*

is a register containing an address to branch to.

### Operation

The BXJ instruction causes a branch to the address contained in *Rm* and changes the instruction set state to Jazelle.

#### Note

In Armv8, BXJ behaves as a BX instruction. This means it causes a branch to an address and instruction set specified by a register.

### Instruction availability and branch ranges

The following table shows the BXJ instructions that are available in A32 and T32 state. Instructions that are not shown in this table are not available.

**Table C2-8 BXJ instruction availability and range**

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
<code>BXJ Rm</code>	Available	-	Available
<code>BXJ{cond} Rm</code>	Available	-	-

### Register restrictions

You can use SP for *Rm* in the BXJ A32 instruction but this is deprecated.

You cannot use SP in the BXJ T32 instruction.

### Condition flags

The BXJ instruction does not change the flags.

### Availability

See the preceding table for details of availability of the BXJ instruction in both instruction sets.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.23 CBZ and CBNZ

Compare and Branch on Zero, Compare and Branch on Non-Zero.

### Syntax

`CBZ{q} Rn, Label`

`CBNZ{q} Rn, Label`

where:

**q** Is an optional instruction width specifier.

**Rn** Is the register holding the operand.

**Label** Is the branch destination.

### Usage

You can use the CBZ or CBNZ instructions to avoid changing the condition flags and to reduce the number of instructions.

Except that it does not change the condition flags, `CBZ Rn, label` is equivalent to:

```
CMP      Rn, #0
BEQ      label
```

Except that it does not change the condition flags, `CBNZ Rn, label` is equivalent to:

```
CMP      Rn, #0
BNE      label
```

### Restrictions

The branch destination must be a multiple of 2 in the range 0 to 126 bytes after the instruction and in the same execution state.

These instructions must not be used inside an IT block.

### Condition flags

These instructions do not change the flags.

### Architectures

These 16-bit instructions are available in Armv7-A T32, Armv8-A T32, and Armv8-M only.

There are no Armv7-A A32, or Armv8-A A32 or 32-bit T32 encodings of these instructions.

### Related reference

[C2.14 B on page C2-182](#)

[C2.27 CMP and CMN on page C2-199](#)

[C2.2 Instruction width specifiers on page C2-161](#)

## C2.24 CDP and CDP2

Coprocessor data operations.

————— Note ————

CDP and CDP2 are not supported in Armv8.

### Syntax

`CDP{cond} coproc, #opcode1, CRd, CRn, CRm{, #opcode2}`

`CDP2{cond} coproc, #opcode1, CRd, CRn, CRm{, #opcode2}`

where:

*cond*

is an optional condition code.

In A32 code, *cond* is not permitted for CDP2.

*coproc*

is the name of the coprocessor the instruction is for. The standard name is *pn*, where *n* is an integer in the range 0-15.

*opcode1*

is a 4-bit coprocessor-specific opcode.

*opcode2*

is an optional 3-bit coprocessor-specific opcode.

*CRd, CRn, CRm*

are coprocessor registers.

### Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

### Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.25 CLREX

Clear Exclusive.

### Syntax

CLREX{*cond*}

where:

*cond*

is an optional condition code.

————— Note ————

*cond* is permitted only in T32 code, using a preceding IT instruction, but this is deprecated in Armv8. This is an unconditional instruction in A32.

### Usage

Use the CLREX instruction to clear the local record of the executing processor that an address has had a request for an exclusive access.

CLREX returns a closely-coupled exclusive access monitor to its open-access state. This removes the requirement for a dummy store to memory.

It is implementation defined whether CLREX also clears the global record of the executing processor that an address has had a request for an exclusive access.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit CLREX instruction in T32.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

#### Related information

[Arm Architecture Reference Manual](#)

## C2.26 CLZ

Count Leading Zeros.

### Syntax

`CLZ{cond} Rd, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*

is the operand register.

### Operation

The `CLZ` instruction counts the number of leading zeros in the value in *Rm* and returns the result in *Rd*. The result value is 32 if no bits are set in the source register, and zero if bit 31 is set.

### Register restrictions

You cannot use PC for any operand.

You can use SP in these A32 instructions but this is deprecated.

You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Examples

```
CLZ    r4,r9
CLZNE  r2,r3
```

Use the `CLZ` T32 instruction followed by a left shift of *Rm* by the resulting *Rd* value to normalize the value of register *Rm*. Use `MOVS`, rather than `MOV`, to flag the case where *Rm* is zero:

```
CLZ r5, r9
MOVS r9, r9, LSL r5
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.27 CMP and CMN

Compare and Compare Negative.

### Syntax

`CMP{cond} Rn, Operand2`

`CMN{cond} Rn, Operand2`

where:

*cond*

is an optional condition code.

*Rn*

is the register holding the first operand.

*Operand2*

is a flexible second operand.

### Operation

These instructions compare the value in a register with *Operand2*. They update the condition flags on the result, but do not place the result in any register.

The `CMP` instruction subtracts the value of *Operand2* from the value in *Rn*. This is the same as a `SUBS` instruction, except that the result is discarded.

The `CMN` instruction adds the value of *Operand2* to the value in *Rn*. This is the same as an `ADDS` instruction, except that the result is discarded.

In certain circumstances, the assembler can substitute `CMN` for `CMP`, or `CMP` for `CMN`. Be aware of this when reading disassembly listings.

### Use of PC in A32 and T32 instructions

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

You can use PC (`R15`) in these A32 instructions without register controlled shift but this is deprecated.

If you use PC as *Rn* in A32 instructions, the value used is the address of the instruction plus 8.

You cannot use PC for any operand in these T32 instructions.

### Use of SP in A32 and T32 instructions

You can use SP for *Rn* in A32 and T32 instructions.

You can use SP for *Rm* in A32 instructions but this is deprecated.

You can use SP for *Rm* in a 16-bit T32 `CMP Rn, Rm` instruction but this is deprecated. Other uses of SP for *Rm* are not permitted in T32.

### Condition flags

These instructions update the N, Z, C and V flags according to the result.

### 16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

**CMP Rn, Rm**

Lo register restriction does not apply.

**CMN Rn, Rm**

*Rn* and *Rm* must both be Lo registers.

**CMP Rn, #imm**

Rn must be a Lo register. imm range 0-255.

### Correct examples

```
CMP      r2, r9
CMN      r0, #6400
CMPGT   sp, r7, LSL #2
```

### Incorrect example

```
CMP      r2, pc, ASR r0 ; PC not permitted with register-controlled
; shift.
```

### Related reference

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.28 CPS

Change Processor State.

### Syntax

`CPSeffect iflags{, #mode}`

`CPS #mode`

where:

`effect`

is one of:

`IE`

Interrupt or abort enable.

`ID`

Interrupt or abort disable.

`iflags`

is a sequence of one or more of:

`a`

Enables or disables imprecise aborts.

`i`

Enables or disables IRQ interrupts.

`f`

Enables or disables FIQ interrupts.

`mode`

specifies the number of the mode to change to.

### Usage

Changes one or more of the mode, A, I, and F bits in the CPSR, without changing the other CPSR bits.

CPS is only permitted in privileged software execution, and has no effect in User mode.

CPS cannot be conditional, and is not permitted in an IT block.

### Condition flags

This instruction does not change the condition flags.

### 16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

- `CPSIE iflags.`
- `CPSID iflags.`

You cannot specify a mode change in a 16-bit T32 instruction.

### Architectures

This instruction is available in A32 and T32.

In T32, 16-bit and 32-bit versions of this instruction are available.

### Examples

```
CPSIE if      ; Enable IRQ and FIQ interrupts.  
CPSID A      ; Disable imprecise aborts.  
CPSID ai, #17 ; Disable imprecise aborts and interrupts, and enter
```

```
CPS #16      ; FIQ mode.  
             ; Enter User mode.
```

### Related concepts

[A2.2 Processor modes, and privileged and unprivileged software execution](#) on page A2-55

## C2.29 CRC32

CRC32 performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register.

### Syntax

```
CRC32B{q} Rd, Rn, Rm ; A1 Wd = CRC32(Wn, Rm[<7:0>])  
CRC32H{q} Rd, Rn, Rm ; A1 Wd = CRC32(Wn, Rm[<15:0>])  
CRC32W{q} Rd, Rn, Rm ; A1 Wd = CRC32(Wn, Rm[<31:0>])  
CRC32B{q} Rd, Rn, Rm ; T1 Wd = CRC32(Wn, Rm[<7:0>])  
CRC32H{q} Rd, Rn, Rm ; T1 Wd = CRC32(Wn, Rm[<15:0>])  
CRC32W{q} Rd, Rn, Rm ; T1 Wd = CRC32(Wn, Rm[<31:0>])
```

Where:

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.  
A CRC32 instruction must be unconditional.

**Rd**

Is the general-purpose accumulator output register.

**Rn**

Is the general-purpose accumulator input register.

**Rm**

Is the general-purpose data source register.

### Architectures supported

Supported in architecture Armv8.1 and later. Optionally supported in the Armv8-A architecture.

### Usage

CRC32 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, or 32 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.

————— Note —————

*ID\_ISAR5.CRC32* indicates whether this instruction is supported in the T32 and A32 instruction sets.

————— Note —————

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[C2.29 CRC32](#) on page C2-203

[C2.1 A32 and T32 instruction summary](#) on page C2-156

## C2.30 CRC32C

CRC32C performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register.

### Syntax

```
CRC32CB{q} Rd, Rn, Rm ; A1 Wd = CRC32C(Wn, Rm[<7:0>])  
CRC32CH{q} Rd, Rn, Rm ; A1 Wd = CRC32C(Wn, Rm[<15:0>])  
CRC32CW{q} Rd, Rn, Rm ; A1 Wd = CRC32C(Wn, Rm[<31:0>])  
CRC32CB{q} Rd, Rn, Rm ; T1 Wd = CRC32C(Wn, Rm[<7:0>])  
CRC32CH{q} Rd, Rn, Rm ; T1 Wd = CRC32C(Wn, Rm[<15:0>])  
CRC32CW{q} Rd, Rn, Rm ; T1 Wd = CRC32C(Wn, Rm[<31:0>])
```

Where:

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.  
A CRC32C instruction must be unconditional.

**Rd**

Is the general-purpose accumulator output register.

**Rn**

Is the general-purpose accumulator input register.

**Rm**

Is the general-purpose data source register.

### Architectures supported

Supported in architecture Armv8-A.1 and later. Optionally supported in the Armv8-A architecture.

### Usage

CRC32C 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, or 32 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.

————— Note —————

*ID\_ISAR5.CRC32* indicates whether this instruction is supported in the T32 and A32 instruction sets.

————— Note —————

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[C2.29 CRC32](#) on page C2-203

[C2.1 A32 and T32 instruction summary](#) on page C2-156

## C2.31 CSDB

Consumption of Speculative Data Barrier.

### Syntax

CSDB{c}{q} ; A32

CSDB{c}.W ; T32

Where:

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**c**

Is an optional condition code. See [Chapter C1 Condition Codes](#) on page C1-133.

### Usage

Consumption of Speculative Data Barrier is a memory barrier that controls Speculative execution and data value prediction. Arm Compiler supports the mitigation of the Variant 1 mechanism that is described in the whitepaper at [Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism](#).

The CSDB instruction allows Speculative execution of:

- Branch instructions.
- Instructions that write to the PC.
- Instructions that are not a result of data value predictions.
- Instructions that are the result of PSTATE.{N,Z,C,V} predictions from conditional branch instructions or from conditional instructions that write to the PC.

The CSDB instruction prevents Speculative execution of:

- Non-branch instructions.
- Instructions that do not write to the PC.
- Instructions that are the result of data value predictions.
- Instructions that are the result of PSTATE.{N,Z,C,V} predictions from instructions other than conditional branch instructions and conditional instructions that write to the PC.

### CONSTRAINED UNPREDICTABLE behavior

For conditional CSDB instructions that specify a condition {c} other than AL in A32, and for any condition {c} used inside an IT block in T32, then how the instructions are rejected depends on your assembler implementation. See your assembler documentation for details.

---

#### Note

---

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

### Examples

The following example shows a code sequence that could result in the processor loading data from an untrusted location that is provided by a user as the result of Speculative execution of instructions:

```
CMP R0, R1
BGE out_of_range
LDRB R4, [R5, R0] ; load data from list A
; speculative execution of this instruction
; must be prevented
AND R4, R4, #1
LSL R4, R4, #8
ADD R4, R4, #0x200
CMP R4, R6
BGE out_of_range
```

```
LDRB R7, [R8, R4] ; load data from list B  
out_of_range
```

In this example:

- There are two list objects A and B.
- A contains a list of values that are used to calculate offsets from which data can be loaded from B.
- R1 is the length of A.
- R5 is the base address of A.
- R6 is the length of B.
- R8 is the base address of B.
- R0 is an untrusted offset that is provided by a user, and is used to load an element from A.

When R0 is greater-than or equal-to the length of A, it is outside the address range of A. Therefore, the first branch instruction BGE `out_of_range` is taken, and instructions LDRB R4, [R5, R0] through LDRB R7, [R8, R4] are skipped.

Without a CSDB instruction, these skipped instructions can still be speculatively executed, and could result in:

- If R0 is maliciously set to an incorrect value, then data can be loaded into R4 from an address outside the address range of A.
- Data can be loaded into R7 from an address outside the address range of B.

To mitigate against these untrusted accesses, add a pair of MOVGE and CSDB instructions between the BGE `out_of_range` and LDRB R4, [R5, R0] instructions as follows:

```
CMP R0, R1  
BGE out_of_range  
  
MOVGE R0, #0      ; conditionally clears the untrusted  
; offset provided by the user so that  
; it cannot affect any other code  
  
CSDB             ; new barrier instruction  
  
LDRB R4, [R5, R0] ; load data from list A  
; speculative execution of this instruction  
; is prevented  
AND R4, R4, #1  
LSL R4, R4, #8  
ADD R4, R4, #0x200  
CMP R4, R6  
BGE out_of_range  
LDRB R7, [R8, R4] ; load data from list B  
out_of_range
```

### Related reference

[C2.1 A32 and T32 instruction summary](#) on page C2-156

[C2.61 MOV](#) on page C2-252

### Related information

[Arm Processor Security Update](#)

[Compiler support for mitigations](#)

## C2.32 DBG

Debug.

### Syntax

`DBG{cond} {option}`

where:

*cond*

is an optional condition code.

*option*

is an optional limitation on the operation of the hint. The range is 0-15.

### Usage

DBG is a hint instruction. It is optional whether it is implemented or not. If it is not implemented, it behaves as a NOP. The assembler produces a diagnostic message if the instruction executes as NOP on the target.

Debug hint provides a hint to a debugger and related tools. See your debugger and related tools documentation to determine the use, if any, of this instruction.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.71 NOP on page C2-266](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.33 DCPS1 (T32 instruction)

Debug switch to exception level 1 (EL1).

————— Note ————

This instruction is supported only in Armv8.

### Syntax

DCPS1

### Usage

This instruction is valid in Debug state only, and is always UNDEFINED in Non-debug state.

DCPS1 targets EL1 and:

- If EL1 is using AArch32, the processing element (PE) enters SVC mode. If EL3 is using AArch32, Secure SVC is an EL3 mode. This means DCPS1 causes the PE to enter EL3.
- If EL1 is using AArch64, the PE enters EL1h, and executes future instructions as A64 instructions.

In Non-debug state, use the SVC instruction to generate a trap to EL1.

### Availability

This 32-bit instruction is available in T32 only.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.148 SVC on page C2-372](#)

[C2.34 DCPS2 \(T32 instruction\) on page C2-209](#)

[C2.35 DCPS3 \(T32 instruction\) on page C2-210](#)

### Related information

[Arm Architecture Reference Manual](#)

## C2.34 DCPS2 (T32 instruction)

Debug switch to exception level 2.

————— Note ————

This instruction is supported only in Armv8.

### Syntax

DCPS2

### Usage

This instruction is valid in Debug state only, and is always UNDEFINED in Non-debug state.

DCPS2 targets EL2 and:

- If EL2 is using AArch32, the PE enters Hyp mode.
- If EL2 is using AArch64, the PE enters EL2h, and executes future instructions as A64 instructions.

In Non-debug state, use the HVC instruction to generate a trap to EL2.

### Availability

This 32-bit instruction is available in T32 only.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.42 HVC on page C2-220](#)

[C2.33 DCPS1 \(T32 instruction\) on page C2-208](#)

[C2.35 DCPS3 \(T32 instruction\) on page C2-210](#)

### Related information

[Arm Architecture Reference Manual](#)

## C2.35 DCPS3 (T32 instruction)

Debug switch to exception level 3.

————— Note ————

This instruction is supported only in Armv8.

### Syntax

DCPS3

### Usage

This instruction is valid in Debug state only, and is always UNDEFINED in Non-debug state.

DCPS3 targets EL3 and:

- If EL3 is using AArch32, the PE enters Monitor mode.
- If EL3 is using AArch64, the PE enters EL3h, and executes future instructions as A64 instructions.

In Non-debug state, use the SMC instruction to generate a trap to EL3.

### Availability

This 32-bit instruction is available in T32 only.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.115 SMC on page C2-323](#)

[C2.34 DCPS2 \(T32 instruction\) on page C2-209](#)

[C2.33 DCPS1 \(T32 instruction\) on page C2-208](#)

### Related information

[Arm Architecture Reference Manual](#)

## C2.36 DMB

Data Memory Barrier.

### Syntax

`DMB{cond} {option}`

where:

*cond*

is an optional condition code.

————— Note —————

*cond* is permitted only in T32 code. This is an unconditional instruction in A32 code.

*option*

is an optional limitation on the operation of the hint. Permitted values are:

**SY**

Full system barrier operation. This is the default and can be omitted.

**LD**

Barrier operation that waits only for loads to complete.

**ST**

Barrier operation that waits only for stores to complete.

**ISH**

Barrier operation only to the inner shareable domain.

**ISHLD**

Barrier operation that waits only for loads to complete, and only applies to the inner shareable domain.

**ISHST**

Barrier operation that waits only for stores to complete, and only to the inner shareable domain.

**NSH**

Barrier operation only out to the point of unification.

**NSHLD**

Barrier operation that waits only for loads to complete and only applies out to the point of unification.

**NSHST**

Barrier operation that waits only for stores to complete and only out to the point of unification.

**OSH**

Barrier operation only to the outer shareable domain.

**OSHLD**

DMB operation that waits only for loads to complete, and only applies to the outer shareable domain.

**OSHST**

Barrier operation that waits only for stores to complete, and only to the outer shareable domain.

————— Note —————

The options LD, ISHLD, NSHLD, and OSHLD are supported only in the Armv8-A and Armv8-R architectures.

## Operation

Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the DMB instruction are observed before any explicit memory accesses that appear in program order after the DMB instruction. It does not affect the ordering of any other instructions executing on the processor.

## Alias

The following alternative values of *option* are supported, but Arm recommends that you do not use them:

- SH is an alias for ISH.
- SHST is an alias for ISHST.
- UN is an alias for NSH.
- UNST is an alias for NSHST.

## Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### *Related reference*

[C1.9 Condition code suffixes on page C1-142](#)

## C2.37 DSB

Data Synchronization Barrier.

### Syntax

`DSB{cond} {option}`

where:

*cond*

is an optional condition code.

————— Note ————

*cond* is permitted only in T32 code. This is an unconditional instruction in A32 code.

*option*

is an optional limitation on the operation of the hint. Permitted values are:

**SY**

Full system barrier operation. This is the default and can be omitted.

**LD**

Barrier operation that waits only for loads to complete.

**ST**

Barrier operation that waits only for stores to complete.

**ISH**

Barrier operation only to the inner shareable domain.

**ISHLD**

Barrier operation that waits only for loads to complete, and only applies to the inner shareable domain.

**ISHST**

Barrier operation that waits only for stores to complete, and only to the inner shareable domain.

**NSH**

Barrier operation only out to the point of unification.

**NSHLD**

Barrier operation that waits only for loads to complete and only applies out to the point of unification.

**NSHST**

Barrier operation that waits only for stores to complete and only out to the point of unification.

**OSH**

Barrier operation only to the outer shareable domain.

**OSHLD**

DMB operation that waits only for loads to complete, and only applies to the outer shareable domain.

**OSHST**

Barrier operation that waits only for stores to complete, and only to the outer shareable domain.

————— Note ————

The options LD, ISHLD, NSHLD, and OSHLD are supported only in the Armv8-A and Armv8-R architectures.

## Operation

Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction executes until this instruction completes. This instruction completes when:

- All explicit memory accesses before this instruction complete.
- All Cache, Branch predictor and TLB maintenance operations before this instruction complete.

## Alias

The following alternative values of *option* are supported for DSB, but Arm recommends that you do not use them:

- SH is an alias for ISH.
- SHST is an alias for ISHST.
- UN is an alias for NSH.
- UNST is an alias for NSHST.

## Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

## Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.38 EOR

Logical Exclusive OR.

### Syntax

**EOR{S}{cond} Rd, Rn, Operand2**

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

**cond**

is an optional condition code.

**Rd**

is the destination register.

**Rn**

is the register holding the first operand.

**Operand2**

is a flexible second operand.

### Operation

The EOR instruction performs bitwise Exclusive OR operations on the values in *Rn* and *Operand2*.

### Use of PC in T32 instructions

You cannot use PC (R15) for *Rd* or any operand in an EOR instruction.

### Use of PC and SP in A32 instructions

You can use PC and SP with the EOR instruction but they are deprecated.

If you use PC as *Rn*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc,lr instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

### Condition flags

If S is specified, the EOR instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

### 16-bit instructions

The following forms of the EOR instruction are available in T32 code, and are 16-bit instructions:

**EORS Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used outside an IT block.

**EOR{cond} Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used inside an IT block.

It does not matter if you specify EOR{S} Rd, Rm, Rd. The instruction is the same.

### Correct examples

```
EORS    r0,r0,r3,ROR r6
EORS    r7, r11, #0x18181818
```

### Incorrect example

```
EORS    r0,pc,r3,ROR r6      ; PC not permitted with register
; controlled shift
```

#### *Related reference*

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.147 SUBS pc, lr on page C2-370](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.39 ERET

Exception Return.

### Syntax

`ERET{cond}`

where:

*cond*

is an optional condition code.

### Usage

In a processor that implements the Virtualization Extensions, you can use ERET to perform a return from an exception taken to Hyp mode.

### Operation

When executed in Hyp mode, ERET loads the PC from ELR\_hyp and loads the CPSR from SPSR\_hyp.

When executed in any other mode, apart from User or System, it behaves as:

- `MOVS PC, LR` in the A32 instruction set.
- `SUBS PC, LR, #0` in the T32 instruction set.

### Notes

You must not use ERET in User or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

ERET is the preferred synonym for `SUBS PC, LR, #0` in the T32 instruction set.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related concepts

[A2.2 Processor modes, and privileged and unprivileged software execution](#) on page A2-55

### Related reference

[C2.61 MOV](#) on page C2-252

[C2.147 SUBS pc, lr](#) on page C2-370

[C1.9 Condition code suffixes](#) on page C1-142

[C2.42 HVC](#) on page C2-220

## C2.40 ESB

Error Synchronization Barrier.

### Syntax

`ESB{c}{q}`

`ESB{c}.W`

Where:

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**c**

Is an optional condition code. See [Chapter C1 Condition Codes](#) on page C1-133.

### Architectures supported

Supported in the Armv8-A and Armv8-R architectures.

### Usage

Error Synchronization Barrier.

### Related reference

[C2.1 A32 and T32 instruction summary](#) on page C2-156

## C2.41 HLT

Halting breakpoint.

— Note —

This instruction is supported only in the Armv8 architecture.

### Syntax

`HLT{Q} #imm`

Where:

**Q**

is an optional suffix. It only has an effect when Halting debug-mode is disabled. In this case, if Q is specified, the instruction behaves as a NOP. If Q is not specified, the instruction is UNDEFINED.

**imm**

is an expression evaluating to an integer in the range:

- 0-65535 (a 16-bit value) in an A32 instruction.
- 0-63 (a 6-bit value) in a 16-bit T32 instruction.

### Usage

The HLT instruction causes the processor to enter Debug state if Halting debug-mode is enabled.

In both A32 state and T32 state, imm is ignored by the Arm hardware. However, a debugger can use it to store additional information about the breakpoint.

HLT is an unconditional instruction. It must not have a condition code in A32 code. In T32 code, the HLT instruction does not require a condition code suffix because it always executes irrespective of its condition code suffix.

### Availability

This instruction is available in A32 and T32.

In T32, it is only available as a 16-bit instruction.

### Related reference

[C2.71 NOP](#) on page C2-266

## C2.42 HVC

Hypervisor Call.

### Syntax

`HVC #imm`

where:

`imm`

is an expression evaluating to an integer in the range 0-65535.

### Operation

In a processor that implements the Virtualization Extensions, the `HVC` instruction causes a Hypervisor Call exception. This means that the processor enters Hyp mode, the CPSR value is saved to the Hyp mode SPSR, and execution branches to the HVC vector.

`HVC` must not be used if the processor is in Secure state, or in User mode in Non-secure state.

`imm` is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

`HVC` cannot be conditional, and is not permitted in an IT block.

### Notes

The `ERET` instruction performs an exception return from Hyp mode.

### Architectures

This 32-bit instruction is available in A32 and T32. It is available in Armv7 architectures that include the Virtualization Extensions.

There is no 16-bit version of this instruction in T32.

### Related concepts

[A2.2 Processor modes, and privileged and unprivileged software execution](#) on page A2-55

### Related reference

[C2.39 ERET](#) on page C2-217

## C2.43 ISB

Instruction Synchronization Barrier.

### Syntax

`ISB{cond} {option}`

where:

*cond*

is an optional condition code.

————— Note ————

*cond* is permitted only in T32 code. This is an unconditional instruction in A32 code.

*option*

is an optional limitation on the operation of the hint. The permitted value is:

**SY**

Full system barrier operation. This is the default and can be omitted.

### Operation

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the `ISB` are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the ASID, or completed TLB maintenance operations, or branch predictor maintenance operations, in addition to all changes to the CP15 registers, executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`.

In addition, the `ISB` instruction ensures that any branches that appear in program order after it are always written into the branch prediction logic with the context that is visible after the `ISB` instruction. This is required to ensure correct execution of the instruction stream.

————— Note ————

When the target architecture is Armv7-M, you cannot use an `ISB` instruction in an IT block, unless it is the last instruction in the block.

### Architectures

This 32-bit instructions are available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.44 IT

The **IT** (If-Then) instruction makes a single following instruction (the *IT block*) conditional. The conditional instruction must be from a restricted set of 16-bit instructions.

### Syntax

**IT cond**

where:

**cond**

specifies the condition for the following instruction.

### Deprecated syntax

**IT{x{y{z}}}{cond}**

where:

**x**

specifies the condition switch for the second instruction in the IT block.

**y**

specifies the condition switch for the third instruction in the IT block.

**z**

specifies the condition switch for the fourth instruction in the IT block.

**cond**

specifies the condition for the first instruction in the IT block.

The condition switches for the second, third, and fourth instructions in the IT block can be either:

**T**

Then. Applies the condition *cond* to the instruction.

**E**

Else. Applies the inverse condition of *cond* to the instruction.

### Usage

The *IT block* can contain between two and four conditional instructions, where the conditions can be all the same, or some of them can be the logical inverse of the others, but this is deprecated in Armv8.

The conditional instruction (including branches, but excluding the **BKPT** instruction) must specify the condition in the *{cond}* part of its syntax.

You are not required to write **IT** instructions in your code, because the assembler generates them for you automatically according to the conditions specified on the following instructions. However, if you do write **IT** instructions, the assembler validates the conditions specified in the **IT** instructions against the conditions specified in the following instructions.

Writing the **IT** instructions ensures that you consider the placing of conditional instructions, and the choice of conditions, in the design of your code.

When assembling to A32 code, the assembler performs the same checks, but does not generate any **IT** instructions.

With the exception of **CMP**, **CMN**, and **TST**, the 16-bit instructions that normally affect the condition flags, do not affect them when used inside an **IT** block.

A BKPT instruction in an IT block is always executed, so it does not require a condition in the `{cond}` part of its syntax. The IT block continues from the next instruction. Using a BKPT or HLT instruction inside an IT block is deprecated.

---

**Note**


---

You can use an IT block for unconditional instructions by using the AL condition.

---

Conditional branches inside an IT block have a longer branch range than those outside the IT block.

**Restrictions**

The following instructions are not permitted in an IT block:

- IT.
- CBZ and CBNZ.
- TBB and TBH.
- CPS, CPSID and CPSIE.
- SETEND.

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC is only permitted in an IT block if it is the last instruction in the block.
- You cannot branch to any instruction in an IT block, unless when returning from an exception handler.
- You cannot use any assembler directives in an IT block.

---

**Note**


---

`armasm` shows a diagnostic message when any of these instructions are used in an IT block.

---

Using any instruction not listed in the following table in an IT block is deprecated. Also, any explicit reference to R15 (the PC) in the IT block is deprecated.

**Table C2-9 Permitted instructions inside an IT block**

16-bit instruction	When deprecated
MOV, MVN	When $Rm$ or $Rd$ is the PC
LDR, LDRB, LDRH, LDRSB, LDRSH	For PC-relative forms
STR, STRB, STRH	-
ADD, ADC, RSB, SBC, SUB	ADD SP, SP, #imm or SUB SP, SP, #imm or when $Rm$ , $Rdn$ or $Rdm$ is the PC
CMP, CMN	When $Rm$ or $Rn$ is the PC
MUL	-
ASR, LSL, LSR, ROR	-
AND, BIC, EOR, ORR, TST	-
BX, BLX	When $Rm$ is the PC

**Condition flags**

This instruction does not change the flags.

## Exceptions

Exceptions can occur between an IT instruction and the corresponding IT block, or within an IT block. This exception results in entry to the appropriate exception handler, with suitable return information in LR and SPSR.

Instructions designed for use as exception returns can be used as normal to return from the exception, and execution of the IT block resumes correctly. This is the only way that a PC-modifying instruction can branch to an instruction in an IT block.

## Availability

This 16-bit instruction is available in T32 only.

In A32 code, IT is a pseudo-instruction that does not generate any code.

There is no 32-bit version of this instruction.

## Correct examples

```
IT      GT
LDRGT  r0, [r1,#4]

IT      EQ
ADDEQ  r0, r1, r2
```

## Incorrect examples

```
IT      NE
ADD   r0,r0,r1 ; syntax error: no condition code used in IT block

ITT    EQ
MOVEQ  r0,r1
ADDEQ  r0,r0,#1 ; IT block covering more than one instruction is deprecated

IT      GT
LDRGT  r0,label ; LDR (PC-relative) is deprecated in an IT block

IT      EQ
ADDEQ  PC,r0     ; ADD is deprecated when Rdn is the PC
```

## C2.45 LDA

Load-Acquire Register.

— Note —

This instruction is supported only in Armv8.

### Syntax

LDA{cond} Rt, [Rn]

LDAB{cond} Rt, [Rn]

LDAH{cond} Rt, [Rn]

where:

*cond*

is an optional condition code.

*Rt*

is the register to load.

*Rn*

is the register on which the memory address is based.

### Operation

LDA loads data from memory. If any loads or stores appear after a load-acquire in program order, then all observers are guaranteed to observe the load-acquire before observing the loads and stores. Loads and stores appearing before a load-acquire are unaffected.

If a store-release follows a load-acquire, each observer is guaranteed to observe them in program order.

There is no requirement that a load-acquire be paired with a store-release.

### Restrictions

The address specified must be naturally aligned, or an alignment fault is generated.

The PC must not be used for *Rt* or *Rn*.

### Availability

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction.

### Related reference

[C2.46 LDAEX on page C2-226](#)

[C2.139 STL on page C2-354](#)

[C2.140 STLEX on page C2-355](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.46 LDAEX

Load-Acquire Register Exclusive.

— Note —

This instruction is supported only in Armv8.

### Syntax

LDAEX{cond} Rt, [Rn]  
LDAEXB{cond} Rt, [Rn]  
LDAEXH{cond} Rt, [Rn]  
LDAEXD{cond} Rt, Rt2, [Rn]

where:

*cond*

is an optional condition code.

*Rt*

is the register to load.

*Rt2*

is the second register for doubleword loads.

*Rn*

is the register on which the memory address is based.

### Operation

LDAEX loads data from memory.

- If the physical address has the Shared TLB attribute, LDAEX tags the physical address as exclusive access for the current processor, and clears any exclusive access tag for this processor for any other physical address.
- Otherwise, it tags the fact that the executing processor has an outstanding tagged physical address.
- If any loads or stores appear after LDAEX in program order, then all observers are guaranteed to observe the LDAEX before observing the loads and stores. Loads and stores appearing before LDAEX are unaffected.

### Restrictions

The PC must not be used for any of *Rt*, *Rt2*, or *Rn*.

For A32 instructions:

- SP can be used but use of SP for any of *Rt*, or *Rt2* is deprecated.
- For LDAEXD, *Rt* must be an even numbered register, and not LR.
- *Rt2* must be  $R(t+1)$ .

For T32 instructions:

- SP can be used for *Rn*, but must not be used for any of *Rt*, or *Rt2*.
- For LDAEXD, *Rt* and *Rt2* must not be the same register.

### Usage

Use LDAEX and STLEX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding LDAEX and STLEX instructions to a minimum.

— **Note** —

The address used in a STLEX instruction must be the same as the address in the most recently executed LDAEX instruction.

## **Availability**

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions.

### **Related reference**

[C2.139 STL on page C2-354](#)

[C2.45 LDA on page C2-225](#)

[C2.140 STLEX on page C2-355](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.47 LDC and LDC2

Transfer Data from memory to Coprocessor.

————— Note —————

LDC2 is not supported in Armv8.

### Syntax

```
op{L}{cond} coproc, CRd, [Rn]  
op{L}{cond} coproc, CRd, [Rn, #{-}offset] ; offset addressing  
op{L}{cond} coproc, CRd, [Rn, #{-}offset]! ; pre-index addressing  
op{L}{cond} coproc, CRd, [Rn], #{-}offset ; post-index addressing  
op{L}{cond} coproc, CRd, Label  
op{L}{cond} coproc, CRd, [Rn], {option}
```

where:

*op*  
is LDC or LDC2.  
*cond*

is an optional condition code.

In A32 code, *cond* is not permitted for LDC2.

**L**

is an optional suffix specifying a long transfer.

*coproc*

is the name of the coprocessor the instruction is for. The standard name is *pn*, where *n* is an integer whose value must be:

- In the range 0 to 15 in Armv7 and earlier.
- 14 in Armv8.

*CRd*

is the coprocessor register to load.

*Rn*

is the register on which the memory address is based. If PC is specified, the value used is the address of the current instruction plus eight.

**-**

is an optional minus sign. If - is present, the offset is subtracted from *Rn*. Otherwise, the offset is added to *Rn*.

*offset*

is an expression evaluating to a multiple of 4, in the range 0 to 1020.

**!**

is an optional suffix. If ! is present, the address including the offset is written back into *Rn*.

*Label*

is a word-aligned PC-relative expression.

*Label* must be within 1020 bytes of the current instruction.

*option*

is a coprocessor option in the range 0-255, enclosed in braces.

## Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

## Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

## Register restrictions

You cannot use PC for Rn in the pre-index and post-index instructions. These are the forms that write back to Rn.

### *Related reference*

[C1.9 Condition code suffixes on page C1-142](#)

## C2.48 LDM

Load Multiple registers.

### Syntax

`LDM{addr_mode}{cond} Rn{!}, reglist{^}`

where:

*addr\_mode*

is any one of the following:

**IA**

Increment address After each transfer. This is the default, and can be omitted.

**IB**

Increment address Before each transfer (A32 only).

**DA**

Decrement address After each transfer (A32 only).

**DB**

Decrement address Before each transfer.

You can also use the stack oriented addressing mode suffixes, for example, when implementing stacks.

*cond*

is an optional condition code.

*Rn*

is the *base register*, the AArch32 register holding the initial address for the transfer. *Rn* must not be PC.

!

is an optional suffix. If ! is present, the final address is written back into *Rn*.

*reglist*

is a list of one or more registers to be loaded, enclosed in braces. It can contain register ranges.

It must be comma separated if it contains more than one register or register range. Any combination of registers R0 to R15 (PC) can be transferred in A32 state, but there are some restrictions in T32 state.

^

is an optional suffix, available in A32 state only. You must not use it in User mode or System mode. It has the following purposes:

- If *reglist* contains the PC (R15), in addition to the normal multiple register transfer, the SPSR is copied into the CPSR. This is for returning from exception handlers. Use this only from exception modes.
- Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.

### Restrictions on reglist in 32-bit T32 instructions

In 32-bit T32 instructions:

- The SP cannot be in the list.
- The PC and LR cannot both be in the list.
- There must be two or more registers in the list.

If you write an LDM instruction with only one register in reglist, the assembler automatically substitutes the equivalent LDR instruction. Be aware of this when comparing disassembly listings with source code.

### Restrictions on reglist in A32 instructions

A32 load instructions can have SP and PC in the *reglist* but these instructions that include SP in the *reglist* or both PC and LR in the *reglist* are deprecated.

## 16-bit instructions

16-bit versions of a subset of these instructions are available in T32 code.

The following restrictions apply to the 16-bit instructions:

- All registers in *reglist* must be Lo registers.
- *Rn* must be a Lo register.
- *addr\_mode* must be omitted (or IA), meaning increment address after each transfer.
- Writeback must be specified for LDM instructions where *Rn* is not in the *reglist*.

In addition, the PUSH and POP instructions are subsets of the STM and LDM instructions and can therefore be expressed using the STM and LDM instructions. Some forms of PUSH and POP are also 16-bit instructions.

## Loading to the PC

A load to the PC causes a branch to the instruction at the address loaded.

Also:

- Bits[1:0] must not be 0b10.
- If bit[0] is 1, execution continues in T32 state.
- If bit[0] is 0, execution continues in A32 state.

## Loading or storing the base register, with writeback

In A32 or 16-bit T32 instructions, if *Rn* is in *reglist*, and writeback is specified with the ! suffix:

- If the instruction is STM{*addr\_mode*}!{*cond*} and *Rn* is the lowest-numbered register in *reglist*, the initial value of *Rn* is stored. These instructions are deprecated.
- Otherwise, the loaded or stored value of *Rn* cannot be relied on, so these instructions are not permitted.

32-bit T32 instructions are not permitted if *Rn* is in *reglist*, and writeback is specified with the ! suffix.

## Correct example

```
LDM      r8,{r0,r2,r9}      ; LDMIA is a synonym for LDM
```

## Incorrect example

```
LDMDA   r2, {}           ; must be at least one register in list
```

## Related reference

[C2.76 POP on page C2-274](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.49 LDR (immediate offset)

Load with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

### Syntax

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]! ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doubleword
LDRD{cond} Rt, Rt2, [Rn, #offset]! ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset ; post-indexed, doubleword
```

where:

*type*

can be any one of:

**B**

unsigned Byte (Zero extend to 32 bits on loads.)

**SB**

signed Byte (LDR only. Sign extend to 32 bits.)

**H**

unsigned Halfword (Zero extend to 32 bits on loads.)

**SH**

signed Halfword (LDR only. Sign extend to 32 bits.)

-

omitted, for Word.

*cond*

is an optional condition code.

*Rt*

is the register to load.

*Rn*

is the register on which the memory address is based.

*offset*

is an offset. If *offset* is omitted, the address is the contents of *Rn*.

*Rt2*

is the additional register to load for doubleword operations.

Not all options are available in every instruction set and architecture.

### Offset ranges and architectures

The following table shows the ranges of offsets and availability of these instructions:

Table C2-10 Offsets and architectures, LDR, word, halfword, and byte

Instruction	Immediate offset	Pre-indexed	Post-indexed
A32, word or byte <sup>h</sup>	-4095 to 4095	-4095 to 4095	-4095 to 4095
A32, signed byte, halfword, or signed halfword	-255 to 255	-255 to 255	-255 to 255
A32, doubleword	-255 to 255	-255 to 255	-255 to 255
T32 32-bit encoding, word, halfword, signed halfword, byte, or signed byte <sup>h</sup>	-255 to 4095	-255 to 255	-255 to 255
T32 32-bit encoding, doubleword	-1020 to 1020 <sup>i</sup>	-1020 to 1020 <sup>i</sup>	-1020 to 1020 <sup>i</sup>

**Table C2-10 Offsets and architectures, LDR, word, halfword, and byte (continued)**

Instruction	Immediate offset	Pre-indexed	Post-indexed
T32 16-bit encoding, word <sup>j</sup>	0 to 124 <sup>i</sup>	Not available	Not available
T32 16-bit encoding, unsigned halfword <sup>j</sup>	0 to 62 <sup>k</sup>	Not available	Not available
T32 16-bit encoding, unsigned byte <sup>j</sup>	0 to 31	Not available	Not available
T32 16-bit encoding, word, Rn is SP <sup>l</sup>	0 to 1020 <sup>i</sup>	Not available	Not available

### Register restrictions

R<sub>n</sub> must be different from R<sub>t</sub> in the pre-index and post-index forms.

### Doubleword register restrictions

R<sub>n</sub> must be different from R<sub>t2</sub> in the pre-index and post-index forms.

For T32 instructions, you must not specify SP or PC for either R<sub>t</sub> or R<sub>t2</sub>.

For A32 instructions:

- R<sub>t</sub> must be an even-numbered register.
- R<sub>t</sub> must not be LR.
- Arm strongly recommends that you do not use R12 for R<sub>t</sub>.
- R<sub>t2</sub> must be R<sub>(t + 1)</sub>.

### Use of PC

In A32 code you can use PC for R<sub>t</sub> in LDR word instructions and PC for R<sub>n</sub> in LDR instructions.

Other uses of PC are not permitted in these A32 instructions.

In T32 code you can use PC for R<sub>t</sub> in LDR word instructions and PC for R<sub>n</sub> in LDR instructions. Other uses of PC in these T32 instructions are not permitted.

### Use of SP

You can use SP for R<sub>n</sub>.

In A32 code, you can use SP for R<sub>t</sub> in word instructions. You can use SP for R<sub>t</sub> in non-word instructions in A32 code but this is deprecated.

In T32 code, you can use SP for R<sub>t</sub> in word instructions only. All other use of SP for R<sub>t</sub> in these instructions are not permitted in T32 code.

### Examples

```
LDR      r8,[r10]          ; loads R8 from the address in R10.
LDRNE   r2,[r5,#960]!     ; (conditionally) loads R2 from a word
                          ; 960 bytes above the address in R5, and
                          ; increments R5 by 960.
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

<sup>h</sup> For word loads, R<sub>t</sub> can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be 0b00. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.

<sup>i</sup> Must be divisible by 4.

<sup>j</sup> R<sub>t</sub> and R<sub>n</sub> must be in the range R0-R7.

<sup>k</sup> Must be divisible by 2.

<sup>l</sup> R<sub>t</sub> must be in the range R0-R7.

## C2.50 LDR (PC-relative)

Load register. The address is an offset from the PC.

### Syntax

`LDR{type}{cond}{.W} Rt, Label`  
`LDRD{cond} Rt, Rt2, Label ; Doubleword`

where:

*type*

can be any one of:

**B**

unsigned Byte (Zero extend to 32 bits on loads.)

**SB**

signed Byte (LDR only. Sign extend to 32 bits.)

**H**

unsigned Halfword (Zero extend to 32 bits on loads.)

**SH**

signed Halfword (LDR only. Sign extend to 32 bits.)

-

omitted, for Word.

*cond*

is an optional condition code.

**.W**

is an optional instruction width specifier.

**Rt**

is the register to load or store.

**Rt2**

is the second register to load or store.

*Label*

is a PC-relative expression.

*Label* must be within a limited distance of the current instruction.

---

### Note

---

Equivalent syntaxes are available for the STR instruction in A32 code but they are deprecated.

---

### Offset range and architectures

The assembler calculates the offset from the PC for you. The assembler generates an error if *Label* is out of range.

The following table shows the possible offsets between the label and the current instruction:

**Table C2-11 PC-relative offsets**

Instruction	Offset range
A32 LDR, LDRB, LDRSB, LDRH, LDRSH <sup>m</sup>	$\pm 4095$
A32 LDRD	$\pm 255$

---

<sup>m</sup> For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be 0b00. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.

**Table C2-11 PC-relative offsets (continued)**

Instruction	Offset range
32-bit T32 LDR, LDRB, LDRSB, LDRH, LDRSH <sup>m</sup>	$\pm 4095$
32-bit T32 LDRD <sup>n</sup>	$\pm 1020$ <sup>o</sup>
16-bit T32 LDR <sup>p</sup>	0-1020 <sup>o</sup>

### LDR (PC-relative) in T32

You can use the `.W` width specifier to force LDR to generate a 32-bit instruction in T32 code. `LDR.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit LDR.

For forward references, LDR without `.W` always generates a 16-bit instruction in T32 code, even if that results in failure for a target that could be reached using a 32-bit T32 LDR instruction.

### Doubleword register restrictions

For 32-bit T32 instructions, you must not specify SP or PC for either `Rt` or `Rt2`.

For A32 instructions:

- `Rt` must be an even-numbered register.
- `Rt` must not be LR.
- Arm strongly recommends that you do not use R12 for `Rt`.
- `Rt2` must be  $R(t + 1)$ .

### Use of SP

In A32 code, you can use SP for `Rt` in LDR word instructions. You can use SP for `Rt` in LDR non-word A32 instructions but this is deprecated.

In T32 code, you can use SP for `Rt` in LDR word instructions only. All other uses of SP in these instructions are not permitted in T32 code.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

<sup>m</sup> For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be `0b00`. In Armv5T and above, bits[1:0] must not be `0b10`, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.  
<sup>n</sup> In Armv7-M, LDRD (PC-relative) instructions must be on a word-aligned address.  
<sup>o</sup> Must be a multiple of 4.  
<sup>p</sup> Rt must be in the range R0-R7. There are no byte, halfword, or doubleword 16-bit instructions.

## C2.51 LDR (register offset)

Load with register offset, pre-indexed register offset, or post-indexed register offset.

### Syntax

```
LDR{type}{cond} Rt, [Rn, ±Rm {, shift}] ; register offset
LDR{type}{cond} Rt, [Rn, ±Rm {, shift}]! ; pre-indexed ; A32 only
LDR{type}{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed ; A32 only
LDRD{cond} Rt, Rt2, [Rn, ±Rm] ; register offset, doubleword ; A32 only
LDRD{cond} Rt, Rt2, [Rn, ±Rm]! ; pre-indexed, doubleword ; A32 only
LDRD{cond} Rt, Rt2, [Rn], ±Rm ; post-indexed, doubleword ; A32 only
```

where:

*type*

can be any one of:

**B**

unsigned Byte (Zero extend to 32 bits on loads.)

**SB**

signed Byte (LDR only. Sign extend to 32 bits.)

**H**

unsigned Halfword (Zero extend to 32 bits on loads.)

**SH**

signed Halfword (LDR only. Sign extend to 32 bits.)

-

omitted, for Word.

*cond*

is an optional condition code.

*Rt*

is the register to load.

*Rn*

is the register on which the memory address is based.

*Rm*

is a register containing a value to be used as the offset. *-Rm* is not permitted in T32 code.

*shift*

is an optional shift.

*Rt2*

is the additional register to load for doubleword operations.

Not all options are available in every instruction set and architecture.

### Offset register and shift options

The following table shows the ranges of offsets and availability of these instructions:

Table C2-12 Options and architectures, LDR (register offsets)

Instruction	$\pm Rm$ <sup>q</sup>	shift		
A32, word or byte <sup>r</sup>	$\pm Rm$	LSL #0-31	LSR #1-32	
		ASR #1-32	ROR #1-31	RRX
A32, signed byte, halfword, or signed halfword	$\pm Rm$	Not available		
A32, doubleword	$\pm Rm$	Not available		
T32 32-bit encoding, word, halfword, signed halfword, byte, or signed byte <sup>r</sup>	$+ Rm$	LSL #0-3		
T32 16-bit encoding, all except doubleword <sup>s</sup>	$+ Rm$	Not available		

### Register restrictions

In the pre-index and post-index forms,  $Rn$  must be different from  $Rt$ .

#### Doubleword register restrictions

For A32 instructions:

- $Rt$  must be an even-numbered register.
- $Rt$  must not be LR.
- Arm strongly recommends that you do not use R12 for  $Rt$ .
- $Rt2$  must be  $R(t + 1)$ .
- $Rm$  must be different from  $Rt$  and  $Rt2$  in LDRD instructions.
- $Rn$  must be different from  $Rt2$  in the pre-index and post-index forms.

### Use of PC

In A32 instructions you can use PC for  $Rt$  in LDR word instructions, and you can use PC for  $Rn$  in LDR instructions with register offset syntax (that is the forms that do not writeback to the  $Rn$ ).

Other uses of PC are not permitted in A32 instructions.

In T32 instructions you can use PC for  $Rt$  in LDR word instructions. Other uses of PC in these T32 instructions are not permitted.

### Use of SP

You can use SP for  $Rn$ .

In A32 code, you can use SP for  $Rt$  in word instructions. You can use SP for  $Rt$  in non-word A32 instructions but this is deprecated.

You can use SP for  $Rm$  in A32 instructions but this is deprecated.

In T32 code, you can use SP for  $Rt$  in word instructions only. All other use of SP for  $Rt$  in these instructions are not permitted in T32 code.

Use of SP for  $Rm$  is not permitted in T32 state.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

<sup>q</sup> Where  $\pm Rm$  is shown, you can use  $-Rm$ ,  $+Rm$ , or  $Rm$ . Where  $+Rm$  is shown, you cannot use  $-Rm$ .

<sup>r</sup> For word loads,  $Rt$  can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be 0b00. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.

<sup>s</sup>  $Rt$ ,  $Rn$ , and  $Rm$  must all be in the range R0-R7.

## C2.52 LDR (register-relative)

Load register. The address is an offset from a base register.

### Syntax

`LDR{type}{cond}{.W} Rt, Label`  
`LDRD{cond} Rt, Rt2, Label ; Doubleword`

where:

*type*

can be any one of:

**B**

unsigned Byte (Zero extend to 32 bits on loads.)

**SB**

signed Byte (LDR only. Sign extend to 32 bits.)

**H**

unsigned Halfword (Zero extend to 32 bits on loads.)

**SH**

signed Halfword (LDR only. Sign extend to 32 bits.)

-

omitted, for Word.

*cond*

is an optional condition code.

**.W**

is an optional instruction width specifier.

**Rt**

is the register to load or store.

**Rt2**

is the second register to load or store.

*Label*

is a symbol defined by the FIELD directive. *Label* specifies an offset from the base register which is defined using the MAP directive.

*Label* must be within a limited distance of the value in the base register.

### Offset range and architectures

The assembler calculates the offset from the base register for you. The assembler generates an error if *Label* is out of range.

The following table shows the possible offsets between the label and the current instruction:

**Table C2-13 Register-relative offsets**

Instruction	Offset range
A32 LDR, LDRB <sup>t</sup>	$\pm 4095$
A32 LDRSB, LDRH, LDRSH	$\pm 255$
A32 LDRD	$\pm 255$
T32, 32-bit LDR, LDRB, LDRSB, LDRH, LDRSH <sup>t</sup>	-255 to 4095

<sup>t</sup> For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be **0b00**. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.

<sup>u</sup> Must be a multiple of 4.

Table C2-13 Register-relative offsets (continued)

Instruction	Offset range
T32, 32-bit LDRD	$\pm 1020$ <sup>u</sup>
T32, 16-bit LDR <sup>v</sup>	0 to 124 <sup>u</sup>
T32, 16-bit LDRH <sup>v</sup>	0 to 62 <sup>w</sup>
T32, 16-bit LDRB <sup>v</sup>	0 to 31
T32, 16-bit LDR, base register is SP <sup>x</sup>	0 to 1020 <sup>u</sup>

### LDR (register-relative) in T32

You can use the `.W` width specifier to force LDR to generate a 32-bit instruction in T32 code. `LDR.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit LDR.

For forward references, LDR without `.W` always generates a 16-bit instruction in T32 code, even if that results in failure for a target that could be reached using a 32-bit T32 LDR instruction.

### Doubleword register restrictions

For 32-bit T32 instructions, you must not specify SP or PC for either `Rt` or `Rt2`.

For A32 instructions:

- `Rt` must be an even-numbered register.
- `Rt` must not be LR.
- Arm strongly recommends that you do not use R12 for `Rt`.
- `Rt2` must be  $R(t + 1)$ .

### Use of PC

You can use PC for `Rt` in word instructions. Other uses of PC are not permitted in these instructions.

### Use of SP

In A32 code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word A32 instructions but this is deprecated.

In T32 code, you can use SP for `Rt` in word instructions only. All other use of SP for `Rt` in these instructions are not permitted in T32 code.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

<sup>v</sup> Rt and base register must be in the range R0-R7.

<sup>w</sup> Must be a multiple of 2.

<sup>x</sup> Rt must be in the range R0-R7.

## C2.53 LDR, unprivileged

Unprivileged load byte, halfword, or word.

### Syntax

`LDR{type}T{cond} Rt, [Rn {, #offset}] ; immediate offset (32-bit T32 encoding only)`

`LDR{type}T{cond} Rt, [Rn] {, #offset} ; post-indexed (A32 only)`

`LDR{type}T{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed (register) (A32 only)`

where:

*type*

can be any one of:

**B**

unsigned Byte (Zero extend to 32 bits on loads.)

**SB**

signed Byte (Sign extend to 32 bits.)

**H**

unsigned Halfword (Zero extend to 32 bits on loads.)

**SH**

signed Halfword (Sign extend to 32 bits.)

-

omitted, for Word.

*cond*

is an optional condition code.

*Rt*

is the register to load.

*Rn*

is the register on which the memory address is based.

*offset*

is an offset. If offset is omitted, the address is the value in *Rn*.

*Rm*

is a register containing a value to be used as the offset. *Rm* must not be PC.

*shift*

is an optional shift.

### Operation

When these instructions are executed by privileged software, they access memory with the same restrictions as they would have if they were executed by unprivileged software.

When executed by unprivileged software these instructions behave in exactly the same way as the corresponding load instruction, for example LDRSBT behaves in the same way as LDRSB.

### Offset ranges and architectures

The following table shows the ranges of offsets and availability of these instructions.

Table C2-14 Offsets and architectures, LDR (User mode)

Instruction	Immediate offset	Post-indexed	±Rm <sup>y</sup>	shift
A32, word or byte	Not available	-4095 to 4095	±Rm	LSL #0-31
				LSR #1-32

<sup>y</sup> You can use -Rm, +Rm, or Rm.

Table C2-14 Offsets and architectures, LDR (User mode) (continued)

Instruction	Immediate offset	Post-indexed	$\pm Rm$	shift
				ASR #1-32
				ROR #1-31
				RRX
A32, signed byte, halfword, or signed halfword	Not available	-255 to 255	$\pm Rm$	Not available
T32, 32-bit encoding, word, halfword, signed halfword, byte, or signed byte	0 to 255	Not available	Not available	Not available

**Related reference**

[C1.9 Condition code suffixes on page C1-142](#)

## C2.54 LDREX

Load Register Exclusive.

### Syntax

LDREX{cond} Rt, [Rn {, #offset}]

LDREXB{cond} Rt, [Rn]

LDREXH{cond} Rt, [Rn]

LDREXD{cond} Rt, Rt2, [Rn]

where:

*cond*

is an optional condition code.

*Rt*

is the register to load.

*Rt2*

is the second register for doubleword loads.

*Rn*

is the register on which the memory address is based.

*offset*

is an optional offset applied to the value in *Rn*. *offset* is permitted only in 32-bit T32 instructions. If *offset* is omitted, an offset of zero is assumed.

### Operation

LDREX loads data from memory.

- If the physical address has the Shared TLB attribute, LDREX tags the physical address as exclusive access for the current processor, and clears any exclusive access tag for this processor for any other physical address.
- Otherwise, it tags the fact that the executing processor has an outstanding tagged physical address.

LDREXB and LDREXH zero extend the value loaded.

### Restrictions

PC must not be used for any of *Rt*, *Rt2*, or *Rn*.

For A32 instructions:

- SP can be used but use of SP for any of *Rt*, or *Rt2* is deprecated.
- For LDREXD, *Rt* must be an even numbered register, and not LR.
- *Rt2* must be *R(t+1)*.
- *offset* is not permitted.

For T32 instructions:

- SP can be used for *Rn*, but must not be used for *Rt* or *Rt2*.
- For LDREXD, *Rt* and *Rt2* must not be the same register.
- The value of *offset* can be any multiple of four in the range 0-1020.

### Usage

Use LDREX and STREX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding LDREX and STREX instructions to a minimum.

— Note —

The address used in a STREX instruction must be the same as the address in the most recently executed LDREX instruction.

## Architectures

These 32-bit instructions are available in A32 and T32.

The LDREXD instruction is not available in the Armv7-M architecture.

There are no 16-bit versions of these instructions in T32.

## Examples

```
MOV r1, #0x1          ; load the 'lock taken' value
try
    LDREX r0, [LockAddr]   ; load the lock value
    CMP r0, #0             ; is the lock free?
    STREXEQ r0, r1, [LockAddr] ; try and claim the lock
    CMPEQ r0, #0           ; did this succeed?
    BNE try                ; no - try again
    ....                  ; yes - we have the lock
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.55 LSL

Logical Shift Left. This instruction is a preferred synonym for `MOV` instructions with shifted register operands.

### Syntax

`LSL{S}{cond} Rd, Rm, Rs`

`LSL{S}{cond} Rd, Rm, #sh`

where:

*S*

is an optional suffix. If *S* is specified, the condition flags are updated on the result of the operation.

*Rd*

is the destination register.

*Rm*

is the register holding the first operand. This operand is shifted left.

*Rs*

is a register holding a shift value to apply to the value in *Rm*. Only the least significant byte is used.

*sh*

is a constant shift. The range of values permitted is 0-31.

### Operation

`LSL` provides the value of a register multiplied by a power of two, inserting zeros into the vacated bit positions.

### Restrictions in T32 code

T32 instructions must not use PC or SP.

You cannot specify zero for the *sh* value in an `LSL` instruction in an IT block.

### Use of SP and PC in A32 instructions

You can use SP in these A32 instructions but this is deprecated.

You cannot use PC in instructions with the `LSL{S}{cond} Rd, Rm, Rs` syntax. You can use PC for *Rd* and *Rm* in the other syntax, but this is deprecated.

If you use PC as *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the *S* suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

#### Note

The A32 instruction `LSLS{cond} pc,Rm,#sh` always disassembles to the preferred form `MOVS{cond} pc,Rm{,shift}`.

#### Caution

Do not use the *S* suffix when using PC as *Rd* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *Rd* or any operand in the LSL instruction if it has a register-controlled shift.

### Condition flags

If S is specified, the LSL instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

### 16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

**LSLS Rd, Rm, #sh**

*Rd* and *Rm* must both be Lo registers. This form can only be used outside an IT block.

**LSL{cond} Rd, Rm, #sh**

*Rd* and *Rm* must both be Lo registers. This form can only be used inside an IT block.

**LSLS Rd, Rd, Rs**

*Rd* and *Rs* must both be Lo registers. This form can only be used outside an IT block.

**LSL{cond} Rd, Rd, Rs**

*Rd* and *Rs* must both be Lo registers. This form can only be used inside an IT block.

### Architectures

This 32-bit instruction is available in A32 and T32.

This 16-bit T32 instruction is available in T32.

### Example

```
LSLS      r1, r2, r3
```

#### Related reference

[C2.61 MOV on page C2-252](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.56 LSR

Logical Shift Right. This instruction is a preferred synonym for MOV instructions with shifted register operands.

### Syntax

```
LSR{S}{cond} Rd, Rm, Rs  
LSR{S}{cond} Rd, Rm, #sh
```

where:

*S*

is an optional suffix. If *S* is specified, the condition flags are updated on the result of the operation.

*Rd*

is the destination register.

*Rm*

is the register holding the first operand. This operand is shifted right.

*Rs*

is a register holding a shift value to apply to the value in *Rm*. Only the least significant byte is used.

*sh*

is a constant shift. The range of values permitted is 1-32.

### Operation

LSR provides the unsigned value of a register divided by a variable power of two, inserting zeros into the vacated bit positions.

### Restrictions in T32 code

T32 instructions must not use PC or SP.

### Use of SP and PC in A32 instructions

You can use SP in these A32 instructions but they are deprecated.

You cannot use PC in instructions with the `LSR{S}{cond} Rd, Rm, Rs` syntax. You can use PC for *Rd* and *Rm* in the other syntax, but this is deprecated.

If you use PC as *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the *S* suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

#### Note

The A32 instruction `LSRS{cond} pc,Rm,#sh` always disassembles to the preferred form `MOVS{cond} pc,Rm{,shift}`.

#### Caution

Do not use the *S* suffix when using PC as *Rd* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *Rd* or any operand in the LSR instruction if it has a register-controlled shift.

## Condition flags

If S is specified, the instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

## 16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

**LSRS Rd, Rm, #sh**

Rd and Rm must both be Lo registers. This form can only be used outside an IT block.

**LSR{cond} Rd, Rm, #sh**

Rd and Rm must both be Lo registers. This form can only be used inside an IT block.

**LSRS Rd, Rd, Rs**

Rd and Rs must both be Lo registers. This form can only be used outside an IT block.

**LSR{cond} Rd, Rd, Rs**

Rd and Rs must both be Lo registers. This form can only be used inside an IT block.

## Architectures

This 32-bit instruction is available in A32 and T32.

This 16-bit T32 instruction is available in T32.

## Example

```
LSR      r4, r5, r6
```

### Related reference

[C2.61 MOV on page C2-252](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.57 MCR and MCR2

Move to Coprocessor from general-purpose register. Depending on the coprocessor, you might be able to specify various additional operations.

———— Note ————

MCR2 is not supported in Armv8.

### Syntax

`MCR{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

`MCR2{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

where:

*cond*

is an optional condition code.

In A32 code, *cond* is not permitted for MCR2.

*coproc*

is the name of the coprocessor the instruction is for. The standard name is *pn*, where *n* is an integer whose value must be:

- In the range 0-15 in Armv7 and earlier.
- 14 or 15 in Armv8.

*opcode1*

is a 3-bit coprocessor-specific opcode.

*opcode2*

is an optional 3-bit coprocessor-specific opcode.

*Rt*

is a general-purpose register. *Rt* must not be PC.

*CRn*, *CRm*

are coprocessor registers.

### Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

### Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.58 MCRR and MCRR2

Move to Coprocessor from two general-purpose registers. Depending on the coprocessor, you might be able to specify various additional operations.

————— Note —————

MCRR2 is not supported in Armv8.

### Syntax

`MCRR{cond} coproc, #opcode, Rt, Rt2, CRn`

`MCRR2{cond} coproc, #opcode, Rt, Rt2, CRn`

where:

*cond*

is an optional condition code.

In A32 code, *cond* is not permitted for MCRR2.

*coproc*

is the name of the coprocessor the instruction is for. The standard name is *pn*, where *n* is an integer whose value must be:

- In the range 0-15 in Armv7 and earlier.
- 14 or 15 in Armv8.

*opcode*

is a 4-bit coprocessor-specific opcode.

*Rt*, *Rt2*

are general-purpose registers. *Rt* and *Rt2* must not be PC.

*CRn*

is a coprocessor register.

### Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

### Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.59 MLA

Multiply-Accumulate with signed or unsigned 32-bit operands, giving the least significant 32 bits of the result.

### Syntax

`MLA{S}{cond} Rd, Rn, Rm, Ra`

where:

`cond`

is an optional condition code.

`S`

is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`Rd`

is the destination register.

`Rn, Rm`

are registers holding the values to be multiplied.

`Ra`

is a register holding the value to be added.

### Operation

The `MLA` instruction multiplies the values from `Rn` and `Rm`, adds the value from `Ra`, and places the least significant 32 bits of the result in `Rd`.

### Register restrictions

You cannot use PC for any register.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

If `S` is specified, the `MLA` instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flag.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Example

```
MLA      r10, r2, r1, r5
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.60 MLS

Multiply-Subtract, with signed or unsigned 32-bit operands, giving the least significant 32 bits of the result.

### Syntax

`MLS{cond} Rd, Rn, Rm, Ra`

where:

*cond*

is an optional condition code.

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

*Rd*

is the destination register.

*Rn, Rm*

are registers holding the values to be multiplied.

*Ra*

is a register holding the value to be subtracted from.

### Operation

The MLS instruction multiplies the values in *Rn* and *Rm*, subtracts the result from the value in *Ra*, and places the least significant 32 bits of the final result in *Rd*.

### Register restrictions

You cannot use PC for any register.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Example

```
MLS      r4, r5, r6, r7
```

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.61 MOV

Move.

### Syntax

`MOV{S}{cond} Rd, Operand2`

`MOV{cond} Rd, #imm16`

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

**cond**

is an optional condition code.

**Rd**

is the destination register.

**Operand2**

is a flexible second operand.

**imm16**

is any value in the range 0-65535.

### Operation

The MOV instruction copies the value of *Operand2* into *Rd*.

In certain circumstances, the assembler can substitute MVN for MOV, or MOV for MVN. Be aware of this when reading disassembly listings.

### Use of PC and SP in 32-bit T32 encodings

You cannot use PC (R15) for *Rd*, or in *Operand2*, in 32-bit T32 MOV instructions. With the following exceptions, you cannot use SP (R13) for *Rd*, or in *Operand2*:

- `MOV{cond}.W Rd, SP`, where *Rd* is not SP.
- `MOV{cond}.W SP, Rm`, where *Rm* is not SP.

### Use of PC and SP in 16-bit T32 encodings

You can use PC or SP in 16-bit T32 `MOV{cond} Rd, Rm` instructions but these instructions in which both *Rd* and *Rm* are SP or PC are deprecated.

You cannot use PC or SP in any other `MOV{S}` 16-bit T32 instructions.

### Use of PC and SP in A32 MOV

You cannot use PC for *Rd* or any operand in any data processing instruction that has a register-controlled shift.

In instructions without register-controlled shift, the use of PC is deprecated except for the following cases:

- `MOVS PC, LR`.
- `MOV PC, Rm` when *Rm* is not PC or SP.
- `MOV Rd, PC` when *Rd* is not PC or SP.

You can use SP for *Rd* or *Rm*. But this is deprecated except for the following cases:

- `MOV SP, Rm` when *Rm* is not PC or SP.
- `MOV Rd, SP` when *Rd* is not PC or SP.

————— Note ————

- You cannot use PC for *Rd* in `MOV Rd, #imm16` if the `#imm16` value is not a permitted *Operand2* value.  
You can use PC in forms with *Operand2* without register-controlled shift.

If you use PC as *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the `SUBS pc, lr` instruction.

### Condition flags

If S is specified, the instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

## 16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

### `MOVS Rd, #imm`

*Rd* must be a Lo register. *imm* range 0-255. This form can only be used outside an IT block.

### `MOV{cond} Rd, #imm`

*Rd* must be a Lo register. *imm* range 0-255. This form can only be used inside an IT block.

### `MOVS Rd, Rm`

*Rd* and *Rm* must both be Lo registers. This form can only be used outside an IT block.

### `MOV{cond} Rd, Rm`

*Rd* or *Rm* can be Lo or Hi registers.

## Availability

These instructions are available in A32 and T32.

In T32, 16-bit and 32-bit versions of these instructions are available.

## Related reference

[C2.3 Flexible second operand \(\*Operand2\*\) on page C2-162](#)

[C2.147 SUBS pc, lr on page C2-370](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.62 MOVT

Move Top.

### Syntax

`MOVT{cond} Rd, #imm16`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*imm16*

is a 16-bit immediate value.

### Usage

MOVT writes *imm16* to *Rd*[31:16], without affecting *Rd*[15:0].

You can generate any 32-bit immediate with a MOV, MOVT instruction pair.

### Register restrictions

You cannot use PC in A32 or T32 instructions.

You can use SP for *Rd* in A32 instructions but this is deprecated.

You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.63 MRC and MRC2

Move to general-purpose register from Coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

————— Note —————

MRC2 is not supported in Armv8.

### Syntax

`MRC{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

`MRC2{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

where:

*cond*

is an optional condition code.

In A32 code, *cond* is not permitted for MRC2.

*coproc*

is the name of the coprocessor the instruction is for. The standard name is *pn*, where *n* is an integer whose value must be:

- In the range 0-15 in Armv7 and earlier.
- 14 or 15 in Armv8.

*opcode1*

is a 3-bit coprocessor-specific opcode.

*opcode2*

is an optional 3-bit coprocessor-specific opcode.

*Rt*

is the general-purpose register. *Rt* must not be PC.

*Rt* can be APSR\_nzcv. This means that the coprocessor executes an instruction that changes the value of the condition flags in the APSR.

*CRn, CRm*

are coprocessor registers.

### Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

### Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.64 MRRC and MRRC2

Move to two general-purpose registers from coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

— Note —

MRRC2 is not supported in Armv8.

### Syntax

MRRC{*cond*} *coproc*, #*opcode*, *Rt*, *Rt2*, *CRm*

MRRC2{*cond*} *coproc*, #*opcode*, *Rt*, *Rt2*, *CRm*

where:

*cond*

is an optional condition code.

In A32 code, *cond* is not permitted for MRRC2.

*coproc*

is the name of the coprocessor the instruction is for. The standard name is *pn*, where *n* is an integer whose value must be:

- In the range 0-15 in Armv7 and earlier.
- 14 or 15 in Armv8.

*opcode*

is a 4-bit coprocessor-specific opcode.

*Rt*, *Rt2*

are general-purpose registers. *Rt* and *Rt2* must not be PC.

*CRm*

is a coprocessor register.

### Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

### Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.65 MRS (PSR to general-purpose register)

Move the contents of a PSR to a general-purpose register.

### Syntax

MRS{cond} Rd, psr

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*psr*

is one of:

**APSR**

on any processor, in any mode.

**CPSR**

deprecated synonym for APSR and for use in Debug state, on any processor except Armv7-M and Armv6-M.

**SPSR**

on any processor, except Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline, in privileged software execution only.

*Mpsr*

on Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline processors only.

*Mpsr*

can be any of: IPSR, EPSR, IEPSR, IAPSR, EAPSR, MSP, PSP, XPSR, PRIMASK, BASEPRI, BASEPRI\_MAX, FAULTMASK, or CONTROL.

### Usage

Use MRS in combination with MSR as part of a read-modify-write sequence for updating a PSR, for example to change processor mode, or to clear the Q flag.

In process swap code, the programmers' model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations make use of MRS/store and load/MSR instruction sequences.

### SPSR

You must not attempt to access the SPSR when the processor is in User or System mode. This is your responsibility. The assembler cannot warn you about this, because it has no information about the processor mode at execution time.

### CPSR

Arm deprecates reading the CPSR endianness bit (E) with an MRS instruction.

The CPSR execution state bits, other than the E bit, can only be read when the processor is in Debug state, halting debug-mode. Otherwise, the execution state bits in the CPSR read as zero.

The condition flags can be read in any mode on any processor. Use APSR if you are only interested in accessing the condition flags in User mode.

### Register restrictions

You cannot use PC for Rd in A32 instructions. You can use SP for Rd in A32 instructions but this is deprecated.

You cannot use PC or SP for Rd in T32 instructions.

## Condition flags

This instruction does not change the flags.

## Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related concepts

[A2.12 Current Program Status Register in AArch32 state on page A2-66](#)

### Related reference

[C2.66 MRS \(system coprocessor register to general-purpose register\) on page C2-259](#)

[C2.67 MSR \(general-purpose register to system coprocessor register\) on page C2-260](#)

[C2.68 MSR \(general-purpose register to PSR\) on page C2-261](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.66 MRS (system coprocessor register to general-purpose register)

Move to general-purpose register from system coprocessor register.

### Syntax

`MRS{cond} Rn, coproc_register`

`MRS{cond} APSR_nzcv, special_register`

where:

*cond*

is an optional condition code.

*coproc\_register*

is the name of the coprocessor register.

*special\_register*

is the name of the coprocessor register that can be written to APSR\_nzcv. This is only possible for the coprocessor register DBGDSCRint.

*Rn*

is the general-purpose register. *Rn* must not be PC.

### Usage

You can use this pseudo-instruction to read CP14 or CP15 coprocessor registers, with the exception of write-only registers. A complete list of the applicable coprocessor register names is in the *Arm®v7-AR Architecture Reference Manual*. For example:

```
MRS R1, SCTRLR ; writes the contents of the CP15 coprocessor
; register SCTRLR into R1
```

### Architectures

This pseudo-instruction is available in Armv7-A and Armv7-R in A32 and 32-bit T32 code.

There is no 16-bit version of this pseudo-instruction in T32.

#### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C2.67 MSR \(general-purpose register to system coprocessor register\) on page C2-260](#)

[C2.68 MSR \(general-purpose register to PSR\) on page C2-261](#)

[C1.9 Condition code suffixes on page C1-142](#)

#### Related information

[Arm Architecture Reference Manual](#)

## C2.67 MSR (general-purpose register to system coprocessor register)

Move to system coprocessor register from general-purpose register.

### Syntax

`MSR{cond} coproc_register, Rn`

where:

*cond*

is an optional condition code.

*coproc\_register*

is the name of the coprocessor register.

*Rn*

is the general-purpose register. *Rn* must not be PC.

### Usage

You can use this pseudo-instruction to write to any CP14 or CP15 coprocessor writable register. A complete list of the applicable coprocessor register names is in the *Arm Architecture Reference Manual*. For example:

```
MSR SCLTR, R1 ; writes the contents of R1 into the CP15
; coprocessor register SCLTR
```

### Availability

This pseudo-instruction is available in A32 and T32.

This pseudo-instruction is available in Armv7-A and Armv7-R in A32 and 32-bit T32 code.

There is no 16-bit version of this pseudo-instruction in T32.

### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C2.66 MRS \(system coprocessor register to general-purpose register\) on page C2-259](#)

[C2.68 MSR \(general-purpose register to PSR\) on page C2-261](#)

[C1.9 Condition code suffixes on page C1-142](#)

[C2.156 SYS on page C2-385](#)

### Related information

[Arm Architecture Reference Manual](#)

## C2.68 MSR (general-purpose register to PSR)

Load an immediate value, or the contents of a general-purpose register, into the specified fields of a Program Status Register (PSR).

### Syntax

`MSR{cond} APSR_flags, Rm`

where:

*cond*

is an optional condition code.

*flags*

specifies the APSR flags to be moved. *flags* can be one or more of:

**nzcvq**

ALU flags field mask, PSR[31:27] (User mode)

**g**

SIMD GE flags field mask, PSR[19:16] (User mode).

*Rm*

is the general-purpose register. *Rm* must not be PC.

### Syntax on architectures other than Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline

`MSR{cond} APSR_flags, #constant`

`MSR{cond} psr_fields, #constant`

`MSR{cond} psr_fields, Rm`

where:

*cond*

is an optional condition code.

*flags*

specifies the APSR flags to be moved. *flags* can be one or more of:

**nzcvq**

ALU flags field mask, PSR[31:27] (User mode)

**g**

SIMD GE flags field mask, PSR[19:16] (User mode).

*constant*

is an expression evaluating to a numeric value. The value must correspond to an 8-bit pattern rotated by an even number of bits within a 32-bit word. Not available in T32.

*Rm*

is the source register. *Rm* must not be PC.

*psr*

is one of:

**CPSR**

for use in Debug state, also deprecated synonym for APSR

**SPSR**

on any processor, in privileged software execution only.

*fields*

specifies the SPSR or CPSR fields to be moved. *fields* can be one or more of:

**c**

control field mask byte, PSR[7:0] (privileged software execution)

x	extension field mask byte, PSR[15:8] (privileged software execution)
s	status field mask byte, PSR[23:16] (privileged software execution)
f	flags field mask byte, PSR[31:24] (privileged software execution).

### Syntax on architectures Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline only

`MSR{cond} psr, Rm`

where:

*cond*

is an optional condition code.

*Rm*

is the source register. *Rm* must not be PC.

*psr*

can be any of: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, XPSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI\_MAX, FAULTMASK, or CONTROL.

### Usage

In User mode:

- Use APSR to access the condition flags, Q, or GE bits.
- Writes to unallocated, privileged or execution state bits in the CPSR are ignored. This ensures that User mode programs cannot change to privileged software execution.

Arm deprecates using MSR to change the endianness bit (E) of the CPSR, in any mode.

You must not attempt to access the SPSR when the processor is in User or System mode.

### Register restrictions

You cannot use PC in A32 instructions. You can use SP for *Rm* in A32 instructions but this is deprecated.

You cannot use PC or SP in T32 instructions.

### Condition flags

This instruction updates the flags explicitly if the APSR\_nzcvq or CPSR\_f field is specified.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C2.66 MRS \(system coprocessor register to general-purpose register\) on page C2-259](#)

[C2.67 MSR \(general-purpose register to system coprocessor register\) on page C2-260](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.69 MUL

Multiply with signed or unsigned 32-bit operands, giving the least significant 32 bits of the result.

### Syntax

**MUL{S}{cond} {Rd}, Rn, Rm**

where:

*cond*

is an optional condition code.

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

**Rd**

is the destination register.

**Rn, Rm**

are registers holding the values to be multiplied.

### Operation

The **MUL** instruction multiplies the values from *Rn* and *Rm*, and places the least significant 32 bits of the result in *Rd*.

### Register restrictions

You cannot use PC for any register.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

If S is specified, the **MUL** instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flag.

### 16-bit instructions

The following forms of the **MUL** instruction are available in T32 code, and are 16-bit instructions:

**MULS Rd, Rn, Rd**

*Rd* and *Rn* must both be Lo registers. This form can only be used outside an IT block.

**MUL{cond} Rd, Rn, Rd**

*Rd* and *Rn* must both be Lo registers. This form can only be used inside an IT block.

There are no other T32 multiply instructions that can update the condition flags.

### Availability

This instruction is available in A32 and T32.

The **MULS** instruction is available in T32 in a 16-bit encoding.

### Examples

```
MUL      r10, r2, r5
MULS    r0, r2, r2
MULLT   r2, r3, r2
```

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.70 MVN

Move Not.

### Syntax

`MVN{S}{cond} Rd, Operand2`

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Operand2*

is a flexible second operand.

### Operation

The MVN instruction takes the value of *Operand2*, performs a bitwise logical NOT operation on the value, and places the result into *Rd*.

In certain circumstances, the assembler can substitute MVN for MOV, or MOV for MVN. Be aware of this when reading disassembly listings.

### Use of PC and SP in 32-bit T32 MVN

You cannot use PC (R15) for *Rd*, or in *Operand2*, in 32-bit T32 MVN instructions. You cannot use SP (R13) for *Rd*, or in *Operand2*.

### Use of PC and SP in 16-bit T32 instructions

You cannot use PC or SP in any MVN{S} 16-bit T32 instructions.

### Use of PC and SP in A32 MVN

You cannot use PC for *Rd* or any operand in any data processing instruction that has a register-controlled shift.

In instructions without register-controlled shift, use of PC is deprecated.

You can use SP for *Rd* or *Rm*, but this is deprecated.

---

#### Note

---

- PC and SP in A32 instructions are deprecated.
- 

If you use PC as *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc,lr instruction.

### Condition flags

If S is specified, the instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

## 16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

### MVNS Rd, Rm

Rd and Rm must both be Lo registers. This form can only be used outside an IT block.

### MVN{cond} Rd, Rm

Rd and Rm must both be Lo registers. This form can only be used inside an IT block.

## Architectures

This instruction is available in A32 and T32.

### Correct example

```
MVNNE    r11, #0xF000000B ; A32 only. This immediate value is not
                           ; available in T32.
```

### Incorrect example

```
MVN      pc,r3,ASR r0      ; PC not permitted with
                           ; register-controlled shift
```

### Related reference

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.147 SUBS pc, lr on page C2-370](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.71 NOP

No Operation.

### Syntax

`NOP{cond}`

where:

*cond*

is an optional condition code.

### Usage

NOP does nothing. If NOP is not implemented as a specific instruction on your target architecture, the assembler treats it as a pseudo-instruction and generates an alternative instruction that does nothing, such as `MOV r0, r0` (A32) or `MOV r8, r8` (T32).

NOP is not necessarily a time-consuming NOP. The processor might remove it from the pipeline before it reaches the execution stage.

You can use NOP for padding, for example to place the following instruction on a 64-bit boundary in A32, or a 32-bit boundary in T32.

### Architectures

This instruction is available in A32 and T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.72 ORN (T32 only)

Logical OR NOT.

### Syntax

ORN{S}{cond} Rd, Rn, Operand2

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Operand2*

is a flexible second operand.

### Operation

The ORN T32 instruction performs an OR operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*.

In certain circumstances, the assembler can substitute ORN for ORR, or ORR for ORN. Be aware of this when reading disassembly listings.

### Use of PC

You cannot use PC (R15) for *Rd* or any operand in the ORN instruction.

### Condition flags

If S is specified, the ORN instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

### Examples

```
ORN      r7, r11, lr, ROR #4
ORNS     r7, r11, lr, ASR #32
```

### Architectures

This 32-bit instruction is available in T32.

There is no A32 or 16-bit T32 ORN instruction.

### Related reference

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.147 SUBS pc, lr on page C2-370](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.73 ORR

Logical OR.

### Syntax

`ORR{S}{cond} Rd, Rn, Operand2`

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

**cond**

is an optional condition code.

**Rd**

is the destination register.

**Rn**

is the register holding the first operand.

**Operand2**

is a flexible second operand.

### Operation

The ORR instruction performs bitwise OR operations on the values in *Rn* and *Operand2*.

In certain circumstances, the assembler can substitute ORN for ORR, or ORR for ORN. Be aware of this when reading disassembly listings.

### Use of PC in 32-bit T32 instructions

You cannot use PC (R15) for *Rd* or any operand with the ORR instruction.

### Use of PC and SP in A32 instructions

You can use PC and SP with the ORR instruction but this is deprecated.

If you use PC as *Rn*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc,lr instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

### Condition flags

If S is specified, the ORR instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

### 16-bit instructions

The following forms of the ORR instruction are available in T32 code, and are 16-bit instructions:

**ORRS Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used outside an IT block.

**ORR{cond} Rd, Rd, Rm**

*Rd* and *Rm* must both be Lo registers. This form can only be used inside an IT block.

It does not matter if you specify ORR{S} *Rd, Rm, Rd*. The instruction is the same.

## Example

```
ORREQ    r2,r0,r5
```

### Related reference

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.147 SUBS pc, lr on page C2-370](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.74 PKHBT and PKHTB

Halfword Packing instructions that combine a halfword from one register with a halfword from another register. One of the operands can be shifted before extraction of the halfword.

### Syntax

`PKHBT{cond} {Rd}, Rn, Rm{, LSL #Leftshift}`

`PKHTB{cond} {Rd}, Rn, Rm{, ASR #rightshift}`

where:

#### PKHBT

Combines bits[15:0] of *Rn* with bits[31:16] of the shifted value from *Rm*.

#### PKHTB

Combines bits[31:16] of *Rn* with bits[15:0] of the shifted value from *Rm*.

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Rm*

is the register holding the second operand.

*Leftshift*

is in the range 0 to 31.

*rightshift*

is in the range 1 to 32.

### Register restrictions

You cannot use PC for any register.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

These instructions do not change the flags.

### Architectures

These instructions are available in A32.

These 32-bit instructions are available T32. For the Armv7-M architecture, they are only available in an Armv7E-M implementation.

There are no 16-bit versions of these instructions in T32.

### Correct examples

```
PKHBT    r0, r3, r5      ; combine the bottom halfword of R3
                  ; with the top halfword of R5
PKHBT    r0, r3, r5, LSL #16 ; combine the bottom halfword of R3
                  ; with the bottom halfword of R5
```

```
PKHTB    r0, r3, r5, ASR #16 ; combine the top halfword of R3  
          ; with the top halfword of R5
```

You can also scale the second operand by using different values of shift.

### Incorrect example

```
PKHBTEQ r4, r5, r1, ASR #8 ; ASR not permitted with PKHBT
```

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.75 PLD, PLDW, and PLI

Preload Data and Preload Instruction allow the processor to signal the memory system that a data or instruction load from an address is likely in the near future.

### Syntax

```
PLtype{cond} [Rn {, #offset}]  
PLtype{cond} [Rn, ±Rm {, shift}]  
PLtype{cond} Label
```

where:

*type*

can be one of:

**D**

Data address.

**DW**

Data address with intention to write.

**I**

Instruction address.

*type* cannot be DW if the syntax specifies *Label*.

*cond*

is an optional condition code.

---

————— Note ————

*cond* is permitted only in T32 code, using a preceding IT instruction, but this is deprecated in the Armv8 architecture. This is an unconditional instruction in A32 code and you must not use *cond*.

---

*Rn*

is the register on which the memory address is based.

*offset*

is an immediate offset. If offset is omitted, the address is the value in *Rn*.

*Rm*

is a register containing a value to be used as the offset.

*shift*

is an optional shift.

*Label*

is a PC-relative expression.

### Range of offsets

The offset is applied to the value in *Rn* before the preload takes place. The result is used as the memory address for the preload. The range of offsets permitted is:

- -4095 to +4095 for A32 instructions.
- -255 to +4095 for T32 instructions, when *Rn* is not PC.
- -4095 to +4095 for T32 instructions, when *Rn* is PC.

The assembler calculates the offset from the PC for you. The assembler generates an error if *Label* is out of range.

### Register or shifted register offset

In A32 code, the value in *Rm* is added to or subtracted from the value in *Rn*. In T32 code, the value in *Rm* can only be added to the value in *Rn*. The result is used as the memory address for the preload.

The range of shifts permitted is:

- LSL #0 to #3 for T32 instructions.
- Any one of the following for A32 instructions:
  - LSL #0 to #31.
  - LSR #1 to #32.
  - ASR #1 to #32.
  - ROR #1 to #31.
  - RRX.

### Address alignment for preloads

No alignment checking is performed for preload instructions.

### Register restrictions

*Rm* must not be PC. For T32 instructions *Rm* must also not be SP.

*Rn* must not be PC for T32 instructions of the syntax `PLtype{cond} [Rn, ±Rm{, #shift}]`.

### Architectures

The PLD instruction is available in A32.

The 32-bit encoding of PLD is available in T32.

PLDW is available only in the Armv7 architecture and above that implement the Multiprocessing Extensions.

PLI is available only in the Armv7 architecture and above.

There are no 16-bit encodings of these instructions in T32.

These are hint instructions, and their implementation is optional. If they are not implemented, they execute as NOPs.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.76 POP

Pop registers off a full descending stack.

### Syntax

`POP{cond} reglist`

where:

*cond*

is an optional condition code.

*reglist*

is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

### Operation

POP is a synonym for LDMIA sp! reglist. POP is the preferred mnemonic.

— Note —

LDM and LDMFD are synonyms of LDMIA.

Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address.

### POP, with reglist including the PC

This instruction causes a branch to the address popped off the stack into the PC. This is usually a return from a subroutine, where the LR was pushed onto the stack at the start of the subroutine.

Also:

- Bits[1:0] must not be 0b10.
- If bit[0] is 1, execution continues in T32 state.
- If bit[0] is 0, execution continues in A32 state.

### T32 instructions

A subset of this instruction is available in the T32 instruction set.

The following restriction applies to the 16-bit POP instruction:

- *reglist* can only include the Lo registers and the PC.

The following restrictions apply to the 32-bit POP instruction:

- *reglist* must not include the SP.
- *reglist* can include either the LR or the PC, but not both.

### Restrictions on reglist in A32 instructions

The A32 POP instruction cannot have SP but can have PC in the *regList*. The instruction that includes both PC and LR in the *reglist* is deprecated.

### Example

```
POP {r0,r10,pc} ; no 16-bit version available
```

### Related reference

[C2.48 LDM on page C2-230](#)

[C2.77 PUSH on page C2-275](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.77 PUSH

Push registers onto a full descending stack.

### Syntax

PUSH{cond} reglist

where:

*cond*

is an optional condition code.

*reglist*

is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

### Operation

PUSH is a synonym for STMDB sp!, reglist. PUSH is the preferred mnemonic.

— Note —

STMFD is a synonym of STMDB.

Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address.

### T32 instructions

The following restriction applies to the 16-bit PUSH instruction:

- *reglist* can only include the Lo registers and the LR.

The following restrictions apply to the 32-bit PUSH instruction:

- *reglist* must not include the SP.
- *reglist* must not include the PC.

### Restrictions on reglist in A32 instructions

The A32 PUSH instruction can have SP and PC in the *reglist* but the instruction that includes SP or PC in the *reglist* is deprecated.

### Examples

```
PUSH    {r0,r4-r7}
PUSH    {r2,lr}
```

### Related reference

[C2.48 LDM on page C2-230](#)

[C2.76 POP on page C2-274](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.78 QADD

Signed saturating addition.

### Syntax

`QADD{cond} {Rd}, Rm, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the registers holding the operands.

### Operation

The QADD instruction adds the values in *Rm* and *Rn*. It saturates the result to the signed range  $-2^{31} \leq x \leq 2^{31}-1$ .

#### Note

All values are treated as two's complement signed integers by this instruction.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an MRS instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

```
QADD    r0, r1, r9
```

### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[A2.10 The Q flag in AArch32 state on page A2-64](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.79 QADD8

Signed saturating parallel byte-wise addition.

### Syntax

`QADD8{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs four signed integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. It saturates the results to the signed range  $-2^7 \leq x \leq 2^7 - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[A2.10 The Q flag in AArch32 state on page A2-64](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.80 QADD16

Signed saturating parallel halfword-wise addition.

### Syntax

`QADD16{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs two signed integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. It saturates the results to the signed range  $-2^{15} \leq x \leq 2^{15} - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[A2.10 The Q flag in AArch32 state on page A2-64](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.81 QASX

Signed saturating parallel add and subtract halfwords with exchange.

### Syntax

`QASX{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. It saturates the results to the signed range  $-2^{15} \leq x \leq 2^{15} - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[A2.10 The Q flag in AArch32 state on page A2-64](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.82 QDADD

Signed saturating Double and Add.

### Syntax

`QDADD{cond} {Rd}, Rm, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

QDADD calculates  $SAT(Rm + SAT(Rn * 2))$ . It saturates the result to the signed range  $-2^{31} \leq x \leq 2^{31}-1$ . Saturation can occur on the doubling operation, on the addition, or on both. If saturation occurs on the doubling but not on the addition, the Q flag is set but the final result is unsaturated.

---

#### Note

---

All values are treated as two's complement signed integers by this instruction.

---

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an MRS instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[A2.10 The Q flag in AArch32 state on page A2-64](#)

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.83 QDSUB

Signed saturating Double and Subtract.

### Syntax

`QDSUB{cond} {Rd}, Rm, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

`QDSUB` calculates  $SAT(Rm - SAT(Rn * 2))$ . It saturates the result to the signed range  $-2^{31} \leq x \leq 2^{31}-1$ . Saturation can occur on the doubling operation, on the subtraction, or on both. If saturation occurs on the doubling but not on the subtraction, the Q flag is set but the final result is unsaturated.

---

#### Note

---

All values are treated as two's complement signed integers by this instruction.

---

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

```
QDSUBLT r9, r0, r1
```

### Related reference

[A2.10 The Q flag in AArch32 state](#) on page A2-64

[C2.65 MRS \(PSR to general-purpose register\)](#) on page C2-257

[C1.9 Condition code suffixes](#) on page C1-142

## C2.84 QSAX

Signed saturating parallel subtract and add halfwords with exchange.

### Syntax

QSAX{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. It saturates the results to the signed range  $-2^{15} \leq x \leq 2^{15} - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[A2.10 The Q flag in AArch32 state on page A2-64](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.85 QSUB

Signed saturating Subtract.

### Syntax

`QSUB{cond} {Rd}, Rm, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

The QSUB instruction subtracts the value in *Rn* from the value in *Rm*. It saturates the result to the signed range  $-2^{31} \leq x \leq 2^{31}-1$ .

---

#### Note

---

All values are treated as two's complement signed integers by this instruction.

---

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an MRS instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[A2.10 The Q flag in AArch32 state](#) on page A2-64

[C2.65 MRS \(PSR to general-purpose register\)](#) on page C2-257

[C1.9 Condition code suffixes](#) on page C1-142

## C2.86 QSUB8

Signed saturating parallel byte-wise subtraction.

### Syntax

`QSUB8{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand and writes the results into the corresponding bytes of the destination. It saturates the results to the signed range  $-2^7 \leq x \leq 2^7 - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[A2.10 The Q flag in AArch32 state on page A2-64](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.87 QSUB16

Signed saturating parallel halfword-wise subtraction.

### Syntax

`QSUB16{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. It saturates the results to the signed range  $-2^{15} \leq x \leq 2^{15} - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[A2.10 The Q flag in AArch32 state on page A2-64](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.88 RBIT

Reverse the bit order in a 32-bit word.

### Syntax

RBIT{cond} Rd, Rn

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the operand.

### Register restrictions

You cannot use PC for any register.

You can use SP in the A32 instruction but this is deprecated. You cannot use SP in the T32 instruction.

### Condition flags

This instruction does not change the flags.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Example

```
RBIT      r7, r8
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.89 REV

Reverse the byte order in a word.

### Syntax

`REV{cond} Rd, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the operand.

### Usage

You can use this instruction to change endianness. REV converts 32-bit big-endian data into little-endian data or 32-bit little-endian data into big-endian data.

### Register restrictions

You cannot use PC for any register.

You can use SP in the A32 instruction but this is deprecated. You cannot use SP in the T32 instruction.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

`REV Rd, Rm`

*Rd* and *Rm* must both be Lo registers.

### Architectures

This instruction is available in A32 and T32.

### Example

```
REV      r3, r7
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.90 REV16

Reverse the byte order in each halfword independently.

### Syntax

`REV16{cond} Rd, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the operand.

### Usage

You can use this instruction to change endianness. REV16 converts 16-bit big-endian data into little-endian data or 16-bit little-endian data into big-endian data.

### Register restrictions

You cannot use PC for any register.

You can use SP in the A32 instruction but this is deprecated. You cannot use SP in the T32 instruction.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

`REV16 Rd, Rm`

*Rd* and *Rm* must both be Lo registers.

### Architectures

This instruction is available in A32 and T32.

### Example

```
REV16    r0, r0
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.91 REVSH

Reverse the byte order in the bottom halfword, and sign extend to 32 bits.

### Syntax

`REVSH{cond} Rd, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the operand.

### Usage

You can use this instruction to change endianness. REVSH converts either:

- 16-bit signed big-endian data into 32-bit signed little-endian data.
- 16-bit signed little-endian data into 32-bit signed big-endian data.

### Register restrictions

You cannot use PC for any register.

You can use SP in the A32 instruction but this is deprecated. You cannot use SP in the T32 instruction.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

`REVSH Rd, Rm`

*Rd* and *Rm* must both be Lo registers.

### Architectures

This instruction is available in A32 and T32.

### Example

```
REVSH    r0, r5      ; Reverse Signed Halfword
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.92 RFE

Return From Exception.

### Syntax

RFE{addr\_mode}{cond} Rn{!}

where:

*addr\_mode*

is any one of the following:

**IA**

Increment address After each transfer (Full Descending stack)

**IB**

Increment address Before each transfer (A32 only)

**DA**

Decrement address After each transfer (A32 only)

**DB**

Decrement address Before each transfer.

If *addr\_mode* is omitted, it defaults to Increment After.

*cond*

is an optional condition code.

————— Note —————

*cond* is permitted only in T32 code, using a preceding **IT** instruction, but this is deprecated in Armv8. This is an unconditional instruction in A32 code.

*Rn*

specifies the base register. *Rn* must not be PC.

!

is an optional suffix. If ! is present, the final address is written back into *Rn*.

### Usage

You can use RFE to return from an exception if you previously saved the return state using the SRS instruction. *Rn* is usually the SP where the return state information was saved.

### Operation

Loads the PC and the CPSR from the address contained in *Rn*, and the following address. Optionally updates *Rn*.

### Notes

RFE writes an address to the PC. The alignment of this address must be correct for the instruction set in use after the exception return:

- For a return to A32, the address written to the PC must be word-aligned.
- For a return to T32, the address written to the PC must be halfword-aligned.
- For a return to Jazelle, there are no alignment restrictions on the address written to the PC.

No special precautions are required in software to follow these rules, if you use the instruction to return after a valid exception entry mechanism.

Where addresses are not word-aligned, RFE ignores the least significant two bits of *Rn*.

The time order of the accesses to individual words of memory generated by RFE is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

Do not use RFE in unprivileged software execution.

## Architectures

This instruction is available in A32.

This 32-bit T32 instruction is available, except in the Armv7-M and Armv8-M Mainline architectures.

There is no 16-bit version of this instruction.

## Example

```
RFE sp!
```

### Related concepts

[A2.2 Processor modes, and privileged and unprivileged software execution](#) on page [A2-55](#)

### Related reference

[C2.132 SRS](#) on page [C2-342](#)

[C1.9 Condition code suffixes](#) on page [C1-142](#)

## C2.93 ROR

Rotate Right. This instruction is a preferred synonym for **MOV** instructions with shifted register operands.

### Syntax

**ROR{S}{cond} Rd, Rm, Rs**  
**ROR{S}{cond} Rd, Rm, #sh**

where:

**S**

is an optional suffix. If **S** is specified, the condition flags are updated on the result of the operation.

**Rd**

is the destination register.

**Rm**

is the register holding the first operand. This operand is shifted right.

**Rs**

is a register holding a shift value to apply to the value in **Rm**. Only the least significant byte is used.

**sh**

is a constant shift. The range of values is 1-31.

### Operation

**ROR** provides the value of the contents of a register rotated by a value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

### Restrictions in T32 code

T32 instructions must not use PC or SP.

### Use of SP and PC in A32 instructions

You can use SP in these A32 instructions but this is deprecated.

You cannot use PC in instructions with the **ROR{S}{cond} Rd, Rm, Rs** syntax. You can use PC for **Rd** and **Rm** in the other syntax, but this is deprecated.

If you use PC as **Rm**, the value used is the address of the instruction plus 8.

If you use PC as **Rd**:

- Execution branches to the address corresponding to the result.
- If you use the **S** suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

#### Note

The A32 instruction **RORS{cond} pc,Rm,#sh** always disassembles to the preferred form **MOVS{cond} pc,Rm{,shift}**.

———— Caution ———

Do not use the S suffix when using PC as *Rd* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *Rd* or any operand in this instruction if it has a register-controlled shift.

**Condition flags**

If S is specified, the instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

**16-bit instructions**

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

**RORS Rd, Rd, Rs**

*Rd* and *Rs* must both be Lo registers. This form can only be used outside an IT block.

**ROR{cond} Rd, Rd, Rs**

*Rd* and *Rs* must both be Lo registers. This form can only be used inside an IT block.

**Architectures**

This instruction is available in A32 and T32.

**Example**

```
ROR      r4, r5, r6
```

**Related reference**

[C2.61 MOV on page C2-252](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.94 RRX

Rotate Right with Extend. This instruction is a preferred synonym for MOV instructions with shifted register operands.

### Syntax

$\text{RRX}\{\text{S}\}\{\text{cond}\} \text{ Rd}, \text{ Rm}$

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

**Rd**

is the destination register.

**Rm**

is the register holding the first operand. This operand is shifted right.

### Operation

RRX provides the value of the contents of a register shifted right one bit. The old carry flag is shifted into bit[31]. If the S suffix is present, the old bit[0] is placed in the carry flag.

### Restrictions in T32 code

T32 instructions must not use PC or SP.

### Use of SP and PC in A32 instructions

You can use SP in this A32 instruction but this is deprecated.

If you use PC as *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

---

#### Note

---

The A32 instruction  $\text{RRXS}\{\text{cond}\} \text{ pc}, \text{Rm}$  always disassembles to the preferred form  $\text{MOVS}\{\text{cond}\} \text{ pc}, \text{Rm}\{\text{,shift}\}$ .

---

---

#### Caution

---

Do not use the S suffix when using PC as *Rd* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

---

You cannot use PC for *Rd* or any operand in this instruction if it has a register-controlled shift.

### Condition flags

If S is specified, the instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

## Architectures

The 32-bit instruction is available in A32 and T32.

There is no 16-bit instruction in T32.

### ***Related reference***

[C2.61 MOV on page C2-252](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.95 RSB

Reverse Subtract without carry.

### Syntax

`RSB{S}{cond} {Rd}, Rn, Operand2`

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Operand2*

is a flexible second operand.

### Operation

The RSB instruction subtracts the value in *Rn* from the value of *Operand2*. This is useful because of the wide range of options for *Operand2*.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

### Use of PC and SP in T32 instructions

You cannot use PC (R15) for *Rd* or any operand.

You cannot use SP (R13) for *Rd* or any operand.

### Use of PC and SP in A32 instructions

You cannot use PC for *Rd* or any operand in an RSB instruction that has a register-controlled shift.

Use of PC for any operand, in instructions without register-controlled shift, is deprecated.

If you use PC (R15) as *Rn* or *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc,lr instruction.

Use of SP and PC in A32 instructions is deprecated.

### Condition flags

If S is specified, the RSB instruction updates the N, Z, C and V flags according to the result.

### 16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

**RSBS Rd, Rn, #0**

Rd and Rn must both be Lo registers. This form can only be used outside an IT block.

**RSB{cond} Rd, Rn, #0**

Rd and Rn must both be Lo registers. This form can only be used inside an IT block.

### Example

```
RSB      r4, r4, #1280      ; subtracts contents of R4 from 1280
```

### Related reference

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.96 RSC

Reverse Subtract with Carry.

### Syntax

RSC{S}{cond} {Rd}, Rn, Operand2

where:

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Operand2*

is a flexible second operand.

### Usage

The RSC instruction subtracts the value in *Rn* from the value of *Operand2*. If the carry flag is clear, the result is reduced by one.

You can use RSC to synthesize multiword arithmetic.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

RSC is not available in T32 code.

### Use of PC and SP

Use of PC and SP is deprecated.

You cannot use PC for *Rd* or any operand in an RSC instruction that has a register-controlled shift.

If you use PC (R15) as *Rn* or *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc,lr instruction.

### Condition flags

If S is specified, the RSC instruction updates the N, Z, C and V flags according to the result.

### Correct example

```
RSCSLE r0,r5,r0,LSL r4 ; conditional, flags set
```

### Incorrect example

```
RSCSLE r0,pc,r0,LSL r4 ; PC not permitted with register
                           ; controlled shift
```

**Related reference**

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.97 SADD8

Signed parallel byte-wise addition.

### Syntax

SADD8{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs four signed integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. The results are modulo 2<sup>8</sup>. It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[0]**

for bits[7:0] of the result.

**GE[1]**

for bits[15:8] of the result.

**GE[2]**

for bits[23:16] of the result.

**GE[3]**

for bits[31:24] of the result.

It sets a GE flag to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to an ADDS instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following SEL instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

[C2.103 SEL on page C2-310](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.98 SADD16

Signed parallel halfword-wise addition.

### Syntax

SADD16{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs two signed integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. The results are modulo  $2^{16}$ . It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[1:0]**

for bits[15:0] of the result.

**GE[3:2]**

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to an ADDS instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following SEL instruction.

---

**Note**

---

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

---

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C2.103 SEL on page C2-310*

*C1.9 Condition code suffixes on page C1-142*

## C2.99 SASX

Signed parallel add and subtract halfwords with exchange.

### Syntax

SASX{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. The results are modulo  $2^{16}$ . It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[1:0]**

for bits[15:0] of the result.

**GE[3:2]**

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to an ADDS or SUBS instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following SEL instruction.

---

— Note —

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

---

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C2.103 SEL* on page C2-310

*C1.9 Condition code suffixes* on page C1-142

## C2.100 SBC

Subtract with Carry.

### Syntax

`SBC{S}{cond} {Rd}, Rn, Operand2`

where:

**S**

is an optional suffix. If **S** is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Operand2*

is a flexible second operand.

### Usage

The **SBC** (Subtract with Carry) instruction subtracts the value of *Operand2* from the value in *Rn*. If the carry flag is clear, the result is reduced by one.

You can use **SBC** to synthesize multiword arithmetic.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

### Use of PC and SP in T32 instructions

You cannot use PC (*R15*) for *Rd*, or any operand.

You cannot use SP (*R13*) for *Rd*, or any operand.

### Use of PC and SP in A32 instructions

You cannot use PC for *Rd* or any operand in an **SBC** instruction that has a register-controlled shift.

Use of PC for any operand in instructions without register-controlled shift, is deprecated.

If you use PC (*R15*) as *Rn* or *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the **S** suffix, see the **SUBS pc,1r** instruction.

Use of SP and PC in **SBC** A32 instructions is deprecated.

### Condition flags

If **S** is specified, the **SBC** instruction updates the N, Z, C and V flags according to the result.

### 16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

**SBCS Rd, Rd, Rm**

Rd and Rm must both be Lo registers. This form can only be used outside an IT block.

**SBC{cond} Rd, Rd, Rm**

Rd and Rm must both be Lo registers. This form can only be used inside an IT block.

**Multiword arithmetic examples**

These instructions subtract one 96-bit integer contained in R9, R10, and R11 from another 96-bit integer contained in R6, R7, and R8, and place the result in R3, R4, and R5:

```
SUBS    r3, r6, r9
SBCS    r4, r7, r10
SBC     r5, r8, r11
```

For clarity, the above examples use consecutive registers for multiword values. There is no requirement to do this. The following, for example, is perfectly valid:

```
SUBS    r6, r6, r9
SBCS    r9, r2, r1
SBC     r2, r8, r11
```

**Related reference**

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.101 SBFX

Signed Bit Field Extract.

### Syntax

`SBFX{cond} Rd, Rn, #lsb, #width`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the source register.

*lsb*

is the bit number of the least significant bit in the bitfield, in the range 0 to 31.

*width*

is the width of the bitfield, in the range 1 to (32–*lsb*).

### Operation

Copies adjacent bits from one register into the least significant bits of a second register, and sign extends to 32 bits.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not alter any flags.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.102 SDIV

Signed Divide.

### Syntax

`SDIV{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the value to be divided.

*Rm*

is a register holding the divisor.

### Register restrictions

PC or SP cannot be used for Rd, Rn, or Rm.

### Architectures

This 32-bit T32 instruction is available in Armv7-R, Armv7-M, and Armv8-M Mainline.

This 32-bit A32 instruction is optional in Armv7-R.

This 32-bit A32 and T32 instruction is available in Armv7-A if Virtualization Extensions are implemented, and optional if not.

There is no 16-bit T32 SDIV instruction.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.103 SEL

Select bytes from each operand according to the state of the APSR GE flags.

### Syntax

`SEL{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Rm*

is the register holding the second operand.

### Operation

The SEL instruction selects bytes from *Rn* or *Rm* according to the APSR GE flags:

- If GE[0] is set, *Rd*[7:0] come from *Rn*[7:0], otherwise from *Rm*[7:0].
- If GE[1] is set, *Rd*[15:8] come from *Rn*[15:8], otherwise from *Rm*[15:8].
- If GE[2] is set, *Rd*[23:16] come from *Rn*[23:16], otherwise from *Rm*[23:16].
- If GE[3] is set, *Rd*[31:24] come from *Rn*[31:24], otherwise from *Rm*[31:24].

### Usage

Use the SEL instruction after one of the signed parallel instructions. You can use this to select maximum or minimum values in multiple byte or halfword data.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Examples

```
SEL      r0, r4, r5
SELLT   r4, r0, r4
```

The following instruction sequence sets each byte in R4 equal to the unsigned minimum of the corresponding bytes of R1 and R2:

```
USUB8   r4, r1, r2
SEL      r4, r2, r1
```

**Related concepts**

[A2.11 Application Program Status Register](#) on page A2-65

**Related reference**

[C1.9 Condition code suffixes](#) on page C1-142

## C2.104 SETEND

Set the endianness bit in the CPSR, without affecting any other bits in the CPSR.

————— Note —————

This instruction is deprecated in Armv8.

### Syntax

SETEND *specifier*

where:

*specifier*

is one of:

**BE**

Big-endian.

**LE**

Little-endian.

### Usage

Use SETEND to access data of different endianness, for example, to access several big-endian DMA-formatted data fields from an otherwise little-endian application.

SETEND cannot be conditional, and is not permitted in an IT block.

### Architectures

This instruction is available in A32 and 16-bit T32.

This 16-bit instruction is available in T32, except in the Armv6-M and Armv7-M architectures.

There is no 32-bit version of this instruction in T32.

### Example

```
SETEND BE      ; Set the CPSR E bit for big-endian accesses
LDR    r0, [r2, #header]
LDR    r1, [r2, #CRC32]
SETEND le      ; Set the CPSR E bit for little-endian accesses
                ; for the rest of the application
```

## C2.105 SETPAN

Set Privileged Access Never.

### Syntax

SETPAN{q} #imm ; A1 general registers (A32)

SETPAN{q} #imm ; T1 general registers (T32)

Where:

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers on page C2-161](#).

**imm**

Is the unsigned immediate 0 or 1.

### Architectures supported

Supported in Armv8.1 and later.

### Usage

Set Privileged Access Never writes a new value to *PSTATE.PAN*.

This instruction is available only in privileged mode and it is a NOP when executed in User mode.

### Related reference

[C2.1 A32 and T32 instruction summary on page C2-156](#)

## C2.106 SEV

Set Event.

### Syntax

`SEV{cond}`

where:

*cond*

is an optional condition code.

### Operation

This is a hint instruction. It is optional whether it is implemented or not. If it is not implemented, it executes as a NOP. The assembler produces a diagnostic message if the instruction executes as a NOP on the target.

SEV causes an event to be signaled to all cores within a multiprocessor system. If SEV is implemented, WFE must also be implemented.

### Availability

This instruction is available in A32 and T32.

#### Related reference

[C2.107 SEVL on page C2-315](#)

[C2.71 NOP on page C2-266](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.107 SEVL

Set Event Locally.

————— Note —————

This instruction is supported only in Armv8.

### Syntax

`SEVL{cond}`

where:

*cond*

is an optional condition code.

### Operation

This is a hint instruction. It is optional whether it is implemented or not. If it is not implemented, it executes as a NOP. `armasm` produces a diagnostic message if the instruction executes as a NOP on the target.

`SEVL` causes an event to be signaled to all cores the current processor. `SEVL` is not required to affect other processors although it is permitted to do so.

### Availability

This instruction is available in A32 and T32.

#### *Related reference*

[C2.106 SEV](#) on page C2-314

[C2.71 NOP](#) on page C2-266

[C1.9 Condition code suffixes](#) on page C1-142

## C2.108 SG

Secure Gateway.

### Syntax

SG

### Usage

Secure Gateway marks a valid branch target for branches from Non-secure code that wants to call Secure code.

## C2.109 SHADD8

Signed halving parallel byte-wise addition.

### Syntax

SHADD8{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs four signed integer additions on the corresponding bytes of the operands, halves the results, and writes the results into the corresponding bytes of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.110 SHADD16

Signed halving parallel halfword-wise addition.

### Syntax

SHADD16{*cond*} {*Rd*}, *Rn*, *Rm*

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*, *Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs two signed integer additions on the corresponding halfwords of the operands, halves the results, and writes the results into the corresponding halfwords of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.111 SHASX

Signed halving parallel add and subtract halfwords with exchange.

### Syntax

SHASX{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It halves the results and writes them into the corresponding halfwords of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.112 SHSAX

Signed halving parallel subtract and add halfwords with exchange.

### Syntax

SHSAX{*cond*} {*Rd*}, *Rn*, *Rm*

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*, *Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It halves the results and writes them into the corresponding halfwords of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.113 SHSUB8

Signed halving parallel byte-wise subtraction.

### Syntax

SHSUB8{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand, halves the results, and writes the results into the corresponding bytes of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.114 SHSUB16

Signed halving parallel halfword-wise subtraction.

### Syntax

SHSUB16{*cond*} {*Rd*}, *Rn*, *Rm*

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*, *Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand, halves the results, and writes the results into the corresponding halfwords of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.115 SMC

Secure Monitor Call.

### Syntax

`SMC{cond} #imm4`

where:

*cond*

is an optional condition code.

*imm4*

is a 4-bit immediate value. This is ignored by the Arm processor, but can be used by the SMC exception handler to determine what service is being requested.

---

#### Note

---

SMC was called SMI in earlier versions of the A32 assembly language. SMI instructions disassemble to SMC, with a comment to say that this was formerly SMI.

---

### Architectures

This 32-bit instruction is available in A32 and T32, if the Arm architecture has the Security Extensions.

There is no 16-bit version of this instruction in T32.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

#### Related information

[Arm Architecture Reference Manual](#)

## C2.116 SMLAxy

Signed Multiply Accumulate, with 16-bit operands and a 32-bit result and accumulator.

### Syntax

`SMLA<x><y>{cond} Rd, Rn, Rm, Ra`

where:

`<x>`

is either B or T. B means use the bottom half (bits [15:0]) of *Rn*, T means use the top half (bits [31:16]) of *Rn*.

`<y>`

is either B or T. B means use the bottom half (bits [15:0]) of *Rm*, T means use the top half (bits [31:16]) of *Rm*.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the values to be multiplied.

`Ra`

is the register holding the value to be added.

### Operation

SMLAxy multiplies the 16-bit signed integers from the selected halves of *Rn* and *Rm*, adds the 32-bit result to the 32-bit value in *Ra*, and places the result in *Rd*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, or V flags.

If overflow occurs in the accumulation, SMLAxy sets the Q flag. To read the state of the Q flag, use an MRS instruction.

---

#### Note

---

SMLAxy never clears the Q flag. To clear the Q flag, use an MSR instruction.

---

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

## Examples

SMLABBNE	r0, r2, r1, r10
SMLABT	r0, r0, r3, r5

### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C2.68 MSR \(general-purpose register to PSR\) on page C2-261](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.117 SMLAD

Dual 16-bit Signed Multiply with Addition of products and 32-bit accumulation.

### Syntax

`SMLAD{X}{cond} Rd, Rn, Rm, Ra`

where:

*cond*

is an optional condition code.

*X*

is an optional parameter. If *X* is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

*Rd*

is the destination register.

*Rn, Rm*

are the registers holding the operands.

*Ra*

is the register holding the accumulate operand.

### Operation

SMLAD multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds both products to the value in *Ra* and stores the sum to *Rd*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

```
SMLADLT    r1, r2, r4, r1
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.118 SMLAL

Signed Long Multiply, with optional Accumulate, with 32-bit operands, and 64-bit result and accumulator.

### Syntax

`SMLAL{S}{cond} RdLo, RdHi, Rn, Rm`

where:

**S**

is an optional suffix available in A32 state only. If S is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*RdLo, RdHi*

are the destination registers. They also hold the accumulating value. *RdLo* and *RdHi* must be different registers

*Rn, Rm*

are general-purpose registers holding the operands.

### Operation

The SMLAL instruction interprets the values from *Rn* and *Rm* as two's complement signed integers. It multiplies these integers, and adds the 64-bit result to the 64-bit signed integer contained in *RdHi* and *RdLo*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

If S is specified, this instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.119 SMLALD

Dual 16-bit Signed Multiply with Addition of products and 64-bit Accumulation.

### Syntax

`SMLALD{X}{cond} RdLo, RdHi, Rn, Rm`

where:

`X`

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

`cond`

is an optional condition code.

`RdLo, RdHi`

are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand. `RdHi` and `RdLo` must be different registers.

`Rn, Rm`

are the general-purpose registers holding the operands.

### Operation

`SMLALD` multiplies the bottom halfword of `Rn` with the bottom halfword of `Rm`, and the top halfword of `Rn` with the top halfword of `Rm`. It then adds both products to the value in `RdLo, RdHi` and stores the sum to `RdLo, RdHi`.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

```
SMLALD      r10, r11, r5, r1
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.120 SMLALxy

Signed Multiply-Accumulate with 16-bit operands and a 64-bit accumulator.

### Syntax

`SMLAL<x><y>{cond} RdLo, RdHi, Rn, Rm`

where:

`<x>`

is either B or T. B means use the bottom half (bits [15:0]) of *Rn*, T means use the top half (bits [31:16]) of *Rn*.

`<y>`

is either B or T. B means use the bottom half (bits [15:0]) of *Rm*, T means use the top half (bits [31:16]) of *Rm*.

`cond`

is an optional condition code.

`RdLo, RdHi`

are the destination registers. They also hold the accumulate value. *RdHi* and *RdLo* must be different registers.

`Rn, Rm`

are the general-purpose registers holding the values to be multiplied.

### Operation

`SMLALxy` multiplies the signed integer from the selected half of *Rm* by the signed integer from the selected half of *Rn*, and adds the 32-bit result to the 64-bit value in *RdHi* and *RdLo*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

————— Note —————

`SMLALxy` cannot raise an exception. If overflow occurs on this instruction, the result wraps round without any warning.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Examples

```
SMLALTB    r2, r3, r7, r1
SMLALBTVS   r0, r1, r9, r2
```

**Related reference**

[C1.9 Condition code suffixes on page C1-142](#)

## C2.121 SMLAWy

Signed Multiply-Accumulate Wide, with one 32-bit and one 16-bit operand, and a 32-bit accumulate value, providing the top 32 bits of the result.

### Syntax

`SMLAW<y>{cond} Rd, Rn, Rm, Ra`

where:

`<y>`

is either B or T. B means use the bottom half (bits [15:0]) of `Rm`, T means use the top half (bits [31:16]) of `Rm`.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the values to be multiplied.

`Ra`

is the register holding the value to be added.

### Operation

`SMLAWy` multiplies the signed 16-bit integer from the selected half of `Rm` by the signed 32-bit integer from `Rn`, adds the top 32 bits of the 48-bit result to the 32-bit value in `Ra`, and places the result in `Rd`.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, or V flags.

If overflow occurs in the accumulation, `SMLAWy` sets the Q flag.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.122 SMLSD

Dual 16-bit Signed Multiply with Subtraction of products and 32-bit accumulation.

### Syntax

`SMLSD{X}{cond} Rd, Rn, Rm, Ra`

where:

*cond*

is an optional condition code.

*X*

is an optional parameter. If *X* is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

*Rd*

is the destination register.

*Rn, Rm*

are the registers holding the operands.

*Ra*

is the register holding the accumulate operand.

### Operation

SMLSD multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, adds the difference to the value in *Ra*, and stores the result to *Rd*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Examples

<code>SMLSD</code>	<code>r1, r2, r0, r7</code>
<code>SMLSxD</code>	<code>r11, r10, r2, r3</code>

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.123 SMLSLD

Dual 16-bit Signed Multiply with Subtraction of products and 64-bit accumulation.

### Syntax

`SMLSLD{X}{cond} RdLo, RdHi, Rn, Rm`

where:

`X`

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

`cond`

is an optional condition code.

`RdLo, RdHi`

are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand. `RdHi` and `RdLo` must be different registers.

`Rn, Rm`

are the general-purpose registers holding the operands.

### Operation

`SMLSLD` multiplies the bottom halfword of `Rn` with the bottom halfword of `Rm`, and the top halfword of `Rn` with the top halfword of `Rm`. It then subtracts the second product from the first, adds the difference to the value in `RdLo, RdHi`, and stores the result to `RdLo, RdHi`.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

```
SMLSLD      r3, r0, r5, r1
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.124 SMMLA

Signed Most significant word Multiply with Accumulation.

### Syntax

`SMMLA{R}{cond} Rd, Rn, Rm, Ra`

where:

`R`

is an optional parameter. If `R` is present, the result is rounded, otherwise it is truncated.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the operands.

`Ra`

is a register holding the value to be added or subtracted from.

### Operation

`SMMLA` multiplies the values from `Rn` and `Rm`, adds the value in `Ra` to the most significant 32 bits of the product, and stores the result in `Rd`.

If the optional `R` parameter is specified, `0x80000000` is added before extracting the most significant 32 bits. This has the effect of rounding the result.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.125 SMMLS

Signed Most significant word Multiply with Subtraction.

### Syntax

`SMMLS{R}{cond} Rd, Rn, Rm, Ra`

where:

`R`

is an optional parameter. If `R` is present, the result is rounded, otherwise it is truncated.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the operands.

`Ra`

is a register holding the value to be added or subtracted from.

### Operation

`SMMLS` multiplies the values from `Rn` and `Rm`, subtracts the product from the value in `Ra` shifted left by 32 bits, and stores the most significant 32 bits of the result in `Rd`.

If the optional `R` parameter is specified, `0x80000000` is added before extracting the most significant 32 bits. This has the effect of rounding the result.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.126 SMMUL

Signed Most significant word Multiply.

### Syntax

`SMMUL{R}{cond} {Rd}, Rn, Rm`

where:

`R`

is an optional parameter. If `R` is present, the result is rounded, otherwise it is truncated.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the operands.

`Ra`

is a register holding the value to be added or subtracted from.

### Operation

`SMMUL` multiplies the 32-bit values from `Rn` and `Rm`, and stores the most significant 32 bits of the 64-bit result to `Rd`.

If the optional `R` parameter is specified, `0x80000000` is added before extracting the most significant 32 bits. This has the effect of rounding the result.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Examples

```
SMMULGE    r6, r4, r3
SMMULR     r2, r2, r2
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.127 SMUAD

Dual 16-bit Signed Multiply with Addition of products, and optional exchange of operand halves.

### Syntax

SMUAD{X}{cond} {Rd}, Rn, Rm

where:

X

is an optional parameter. If X is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond

is an optional condition code.

Rd

is the destination register.

Rn, Rm

are the registers holding the operands.

### Operation

SMUAD multiplies the bottom halfword of Rn with the bottom halfword of Rm, and the top halfword of Rn with the top halfword of Rm. It then adds the products and stores the sum to Rd.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

The SMUAD instruction sets the Q flag if the addition overflows.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Examples

SMUAD r2, r3, r2

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.128 SMULxy

Signed Multiply, with 16-bit operands and a 32-bit result.

### Syntax

`SMUL<x><y>{cond} {Rd}, Rn, Rm`

where:

`<x>`

is either B or T. B means use the bottom half (bits [15:0]) of *Rn*, T means use the top half (bits [31:16]) of *Rn*.

`<y>`

is either B or T. B means use the bottom half (bits [15:0]) of *Rm*, T means use the top half (bits [31:16]) of *Rm*.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the values to be multiplied.

### Operation

SMULxy multiplies the 16-bit signed integers from the selected halves of *Rn* and *Rm*, and places the 32-bit result in *Rd*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

These instructions do not affect the N, Z, C, or V flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Examples

```
SMULTBEQ    r8, r7, r9
```

### Related reference

[C2.65 MRS \(PSR to general-purpose register\)](#) on page C2-257

[C2.68 MSR \(general-purpose register to PSR\)](#) on page C2-261

[C1.9 Condition code suffixes](#) on page C1-142

## C2.129 SMULL

Signed Long Multiply, with 32-bit operands and 64-bit result.

### Syntax

`SMULL{S}{cond} RdLo, RdHi, Rn, Rm`

where:

**S**

is an optional suffix available in A32 state only. If S is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*RdLo, RdHi*

are the destination registers. *RdLo* and *RdHi* must be different registers

*Rn, Rm*

are general-purpose registers holding the operands.

### Operation

The SMULL instruction interprets the values from *Rn* and *Rm* as two's complement signed integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

If S is specified, this instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.130 SMULWY

Signed Multiply Wide, with one 32-bit and one 16-bit operand, providing the top 32 bits of the result.

### Syntax

`SMULW<y>{cond} {Rd}, Rn, Rm`

where:

`<y>`

is either B or T. B means use the bottom half (bits [15:0]) of *Rm*, T means use the top half (bits [31:16]) of *Rm*.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the values to be multiplied.

### Operation

`SMULWY` multiplies the signed integer from the selected half of *Rm* by the signed integer from *Rn*, and places the upper 32-bits of the 48-bit result in *Rd*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, or V flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.131 SMUSD

Dual 16-bit Signed Multiply with Subtraction of products, and optional exchange of operand halves.

### Syntax

`SMUSD{X}{cond} {Rd}, Rn, Rm`

where:

`X`

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the operands.

### Operation

`SMUSD` multiplies the bottom halfword of `Rn` with the bottom halfword of `Rm`, and the top halfword of `Rn` with the top halfword of `Rm`. It then subtracts the second product from the first, and stores the difference to `Rd`.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

```
SMUSDXNE    r0, r1, r2
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.132 SRS

Store Return State onto a stack.

### Syntax

```
SRS{addr_mode}{cond} sp{!}, #modenum  
SRS{addr_mode}{cond} #modenum{!} ; This is pre-UAL syntax
```

where:

*addr\_mode*

is any one of the following:

**IA**

Increment address After each transfer

**IB**

Increment address Before each transfer (A32 only)

**DA**

Decrement address After each transfer (A32 only)

**DB**

Decrement address Before each transfer (Full Descending stack).

If *addr\_mode* is omitted, it defaults to Increment After. You can also use stack oriented addressing mode suffixes, for example, when implementing stacks.

*cond*

is an optional condition code.

---

————— Note —————

*cond* is permitted only in T32 code, using a preceding **IT** instruction, but this is deprecated in the Armv8 architecture. This is an unconditional instruction in A32.

---

!

is an optional suffix. If ! is present, the final address is written back into the SP of the mode specified by *modenum*.

*modenum*

specifies the number of the mode whose banked SP is used as the base register. You must use only the defined mode numbers.

### Operation

SRS stores the LR and the SPSR of the current mode, at the address contained in SP of the mode specified by *modenum*, and the following word respectively. Optionally updates SP of the mode specified by *modenum*. This is compatible with the normal use of the **STM** instruction for stack accesses.

---

————— Note —————

For full descending stack, you must use **SRSFD** or **SRSDB**.

---

### Usage

You can use SRS to store return state for an exception handler on a different stack from the one automatically selected.

## Notes

Where addresses are not word-aligned, SRS ignores the least significant two bits of the specified address.

The time order of the accesses to individual words of memory generated by SRS is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

Do not use SRS in User and System modes because these modes do not have a SPSR.

SRS is not permitted in a non-secure state if *modenum* specifies monitor mode.

## Availability

This 32-bit instruction is available in A32 and T32.

The 32-bit T32 instruction is not available in the Armv7-M architecture.

There is no 16-bit version of this instruction in T32.

## Example

```
R13_usr EQU 16
SRSFD sp,#R13_usr
```

### Related concepts

[A2.2 Processor modes, and privileged and unprivileged software execution](#) on page A2-55

### Related reference

[C2.48 LDM](#) on page C2-230

[C1.9 Condition code suffixes](#) on page C1-142

## C2.133 SSAT

Signed Saturate to any bit position, with optional shift before saturating.

### Syntax

`SSAT{cond} Rd, #sat, Rm{, shift}`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*sat*

specifies the bit position to saturate to, in the range 1 to 32.

*Rm*

is the register containing the operand.

*shift*

is an optional shift. It must be one of the following:

**ASR #n**

where *n* is in the range 1-32 (A32) or 1-31 (T32)

**LSL #n**

where *n* is in the range 0-31.

### Operation

The SSAT instruction applies the specified shift, then saturates a signed value to the signed range  $-2^{\text{sat}-1} \leq x \leq 2^{\text{sat}-1} - 1$ .

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an MRS instruction.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Example

```
SSAT      r7, #16, r7, LSL #4
```

### Related reference

[C2.134 SSAT16 on page C2-345](#)

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.134 SSAT16

Parallel halfword Saturate.

### Syntax

`SSAT16{cond} Rd, #sat, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*sat*

specifies the bit position to saturate to, in the range 1 to 16.

*Rn*

is the register holding the operand.

### Operation

Halfword-wise signed saturation to any bit position.

The SSAT16 instruction saturates each signed halfword to the signed range  $-2^{\text{sat}-1} \leq x \leq 2^{\text{sat}-1} - 1$ .

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

If saturation occurs on either halfword, this instruction sets the Q flag. To read the state of the Q flag, use an MRS instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Correct example

```
SSAT16 r7, #12, r7
```

### Incorrect example

```
SSAT16 r1, #16, r2, LSL #4 ; shifts not permitted with halfword  
; saturations
```

### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.135 SSAX

Signed parallel subtract and add halfwords with exchange.

### Syntax

SSAX{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. The results are modulo  $2^{16}$ . It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[1:0]**

for bits[15:0] of the result.

**GE[3:2]**

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to an ADDS or SUBS instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following SEL instruction.

---

**Note**

---

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

---

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C2.103 SEL* on page C2-310

*C1.9 Condition code suffixes* on page C1-142

## C2.136 SSUB8

Signed parallel byte-wise subtraction.

### Syntax

SSUB8{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand and writes the results into the corresponding bytes of the destination. The results are modulo 2<sup>8</sup>. It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[0]**

for bits[7:0] of the result.

**GE[1]**

for bits[15:8] of the result.

**GE[2]**

for bits[23:16] of the result.

**GE[3]**

for bits[31:24] of the result.

It sets a GE flag to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to a SUBS instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following SEL instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

[C2.103 SEL on page C2-310](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.137 SSUB16

Signed parallel halfword-wise subtraction.

### Syntax

SSUB16{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. The results are modulo  $2^{16}$ . It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[1:0]**

for bits[15:0] of the result.

**GE[3:2]**

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to a SUBS instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following SEL instruction.

---

**Note**

---

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

---

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C2.103 SEL on page C2-310*

*C1.9 Condition code suffixes on page C1-142*

## C2.138 STC and STC2

Transfer Data between memory and Coprocessor.

————— Note —————

STC2 is not supported in Armv8.

### Syntax

*op{L}{cond} coproc, CRd, [Rn]*  
*op{L}{cond} coproc, CRd, [Rn, #{-}offset] ; offset addressing*  
*op{L}{cond} coproc, CRd, [Rn, #{-}offset]! ; pre-index addressing*  
*op{L}{cond} coproc, CRd, [Rn], #{-}offset ; post-index addressing*  
*op{L}{cond} coproc, CRd, [Rn], {option}*

where:

*op*

is one of STC or STC2.

*cond*

is an optional condition code.

In A32 code, *cond* is not permitted for STC2.

*L*

is an optional suffix specifying a long transfer.

*coproc*

is the name of the coprocessor the instruction is for. The standard name is *pn*, where *n* is an integer whose value must be:

- In the range 0-15 in Armv7 and earlier.
- 14 in Armv8.

*CRd*

is the coprocessor register to store.

*Rn*

is the register on which the memory address is based. If PC is specified, the value used is the address of the current instruction plus eight.

-

is an optional minus sign. If - is present, the offset is subtracted from *Rn*. Otherwise, the offset is added to *Rn*.

*offset*

is an expression evaluating to a multiple of 4, in the range 0 to 1020.

!

is an optional suffix. If ! is present, the address including the offset is written back into *Rn*.

*option*

is a coprocessor option in the range 0-255, enclosed in braces.

## Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

## Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

## Register restrictions

You cannot use PC for  $Rn$  in the pre-index and post-index instructions. These are the forms that write back to  $Rn$ .

You cannot use PC for  $Rn$  in T32 STC and STC2 instructions.

A32 STC and STC2 instructions where  $Rn$  is PC, are deprecated.

## Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.139 STL

Store-Release Register.

— Note —

This instruction is supported only in Armv8.

### Syntax

`STL{cond} Rt, [Rn]`

`STLB{cond} Rt, [Rn]`

`STLH{cond} Rt, [Rn]`

where:

*cond*

is an optional condition code.

*Rt*

is the register to store.

*Rn*

is the register on which the memory address is based.

### Operation

STL stores data to memory. If any loads or stores appear before a store-release in program order, then all observers are guaranteed to observe the loads and stores before observing the store-release. Loads and stores appearing after a store-release are unaffected.

If a store-release follows a load-acquire, each observer is guaranteed to observe them in program order.

There is no requirement that a store-release be paired with a load-acquire.

All store-release operations are multi-copy atomic, meaning that in a multiprocessor system, if one observer observes a write to memory because of a store-release operation, then all observers observe it. Also, all observers observe all such writes to the same location in the same order.

### Restrictions

The address specified must be naturally aligned, or an alignment fault is generated.

The PC must not be used for *Rt* or *Rn*.

### Availability

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction.

### Related reference

[C2.46 LDAEX on page C2-226](#)

[C2.45 LDA on page C2-225](#)

[C2.140 STLEX on page C2-355](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.140 STLEX

Store-Release Register Exclusive.

— Note —

This instruction is supported only in Armv8.

### Syntax

`STLEX{cond} Rd, Rt, [Rn]`  
`STLEXB{cond} Rd, Rt, [Rn]`  
`STLEXH{cond} Rd, Rt, [Rn]`  
`STLEXD{cond} Rd, Rt, Rt2, [Rn]`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register for the returned status.

*Rt*

is the register to load or store.

*Rt2*

is the second register for doubleword loads or stores.

*Rn*

is the register on which the memory address is based.

### Operation

STLEX performs a conditional store to memory. The conditions are as follows:

- If the physical address does not have the Shared TLB attribute, and the executing processor has an outstanding tagged physical address, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding tagged physical address, the store does not take place, and the value 1 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place, and the value 1 is returned in *Rd*.

If any loads or stores appear before STLEX in program order, then all observers are guaranteed to observe the loads and stores before observing the store-release. Loads and stores appearing after STLEX are unaffected.

All store-release operations are multi-copy atomic.

### Restrictions

The PC must not be used for any of *Rd*, *Rt*, *Rt2*, or *Rn*.

For STLEX, *Rd* must not be the same register as *Rt*, *Rt2*, or *Rn*.

For A32 instructions:

- SP can be used but use of SP for any of *Rd*, *Rt*, or *Rt2* is deprecated.
- For STLEXD, *Rt* must be an even numbered register, and not LR.
- *Rt2* must be *R(t+1)*.

For T32 instructions, SP can be used for *Rn*, but must not be used for any of *Rd*, *Rt*, or *Rt2*.

## Usage

Use LDAEX and STLEX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding LDAEX and STLEX instructions to a minimum.

### Note

The address used in a STLEX instruction must be the same as the address in the most recently executed LDAEX instruction.

## Availability

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions.

### Related reference

[C2.46 LDAEX on page C2-226](#)

[C2.139 STL on page C2-354](#)

[C2.45 LDA on page C2-225](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.141 STM

Store Multiple registers.

### Syntax

`STM{addr_mode}{cond} Rn{!}, reglist{^}`

where:

*addr\_mode*

is any one of the following:

**IA**

Increment address After each transfer. This is the default, and can be omitted.

**IB**

Increment address Before each transfer (A32 only).

**DA**

Decrement address After each transfer (A32 only).

**DB**

Decrement address Before each transfer.

You can also use the stack-oriented addressing mode suffixes, for example when implementing stacks.

*cond*

is an optional condition code.

*Rn*

is the *base register*, the general-purpose register holding the initial address for the transfer. *Rn* must not be PC.

!

is an optional suffix. If ! is present, the final address is written back into *Rn*.

*reglist*

is a list of one or more registers to be stored, enclosed in braces. It can contain register ranges. It must be comma-separated if it contains more than one register or register range. Any combination of registers R0 to R15 (PC) can be transferred in A32 state, but there are some restrictions in T32 state.

^

is an optional suffix, available in A32 state only. You must not use it in User mode or System mode. Data is transferred into or out of the User mode registers instead of the current mode registers.

### Restrictions on reglist in 32-bit T32 instructions

In 32-bit T32 instructions:

- The SP cannot be in the list.
- The PC cannot be in the list.
- There must be two or more registers in the list.

If you write an STM instruction with only one register in *reglist*, the assembler automatically substitutes the equivalent STR instruction. Be aware of this when comparing disassembly listings with source code.

## Restrictions on reglist in A32 instructions

A32 store instructions can have SP and PC in the *reglist* but these instructions that include SP or PC in the *reglist* are deprecated.

### 16-bit instruction

A 16-bit version of this instruction is available in T32 code.

The following restrictions apply to the 16-bit instruction:

- All registers in *reglist* must be Lo registers.
- *Rn* must be a Lo register.
- *addr\_mode* must be omitted (or IA), meaning increment address after each transfer.
- Writeback must be specified for STM instructions.

— Note —

16-bit T32 STM instructions with writeback that specify *Rn* as the lowest register in the *reglist* are deprecated.

In addition, the PUSH and POP instructions are subsets of the STM and LDM instructions and can therefore be expressed using the STM and LDM instructions. Some forms of PUSH and POP are also 16-bit instructions.

### Storing the base register, with writeback

In A32 or 16-bit T32 instructions, if *Rn* is in *reglist*, and writeback is specified with the ! suffix:

- If the instruction is STM{*addr\_mode*}{'cond'} and *Rn* is the lowest-numbered register in *reglist*, the initial value of *Rn* is stored. These instructions are deprecated.
- Otherwise, the stored value of *Rn* cannot be relied on, so these instructions are not permitted.

32-bit T32 instructions are not permitted if *Rn* is in *reglist*, and writeback is specified with the ! suffix.

### Correct example

```
STMDB    r1!,{r3-r6,r11,r12}
```

### Incorrect example

```
STM      r5!,{r5,r4,r9} ; value stored for R5 unknown
```

### Related reference

[C2.76 POP on page C2-274](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.142 STR (immediate offset)

Store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

### Syntax

```
STR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
STR{type}{cond} Rt, [Rn, #offset]! ; pre-indexed
STR{type}{cond} Rt, [Rn], #offset ; post-indexed
STRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doubleword
STRD{cond} Rt, Rt2, [Rn, #offset]! ; pre-indexed, doubleword
STRD{cond} Rt, Rt2, [Rn], #offset ; post-indexed, doubleword
```

where:

*type*

can be any one of:

**B**

Byte

**H**

Halfword

-

omitted, for Word.

*cond*

is an optional condition code.

*Rt*

is the general-purpose register to store.

*Rn*

is the general-purpose register on which the memory address is based.

*offset*

is an offset. If *offset* is omitted, the address is the contents of *Rn*.

*Rt2*

is the additional register to store for doubleword operations.

Not all options are available in every instruction set and architecture.

### Offset ranges and architectures

The following table shows the ranges of offsets and availability of this instruction:

Table C2-15 Offsets and architectures, STR, word, halfword, and byte

Instruction	Immediate offset	Pre-indexed	Post-indexed
A32, word or byte	-4095 to 4095	-4095 to 4095	-4095 to 4095
A32, halfword	-255 to 255	-255 to 255	-255 to 255

**Table C2-15 Offsets and architectures, STR, word, halfword, and byte (continued)**

Instruction	Immediate offset	Pre-indexed	Post-indexed
A32, doubleword	-255 to 255	-255 to 255	-255 to 255
T32 32-bit encoding, word, halfword, or byte	-255 to 4095	-255 to 255	-255 to 255
T32 32-bit encoding, doubleword	-1020 to 1020 <sup>z</sup>	-1020 to 1020 <sup>z</sup>	-1020 to 1020 <sup>z</sup>
T32 16-bit encoding, word <sup>aa</sup>	0 to 124 <sup>z</sup>	Not available	Not available
T32 16-bit encoding, halfword <sup>aa</sup>	0 to 62 <sup>ac</sup>	Not available	Not available
T32 16-bit encoding, byte <sup>aa</sup>	0 to 31	Not available	Not available
T32 16-bit encoding, word, Rn is SP <sup>ab</sup>	0 to 1020 <sup>z</sup>	Not available	Not available

### Register restrictions

R<sub>n</sub> must be different from R<sub>t</sub> in the pre-index and post-index forms.

### Doubleword register restrictions

R<sub>n</sub> must be different from R<sub>t2</sub> in the pre-index and post-index forms.

For T32 instructions, you must not specify SP or PC for either R<sub>t</sub> or R<sub>t2</sub>.

For A32 instructions:

- R<sub>t</sub> must be an even-numbered register.
- R<sub>t</sub> must not be LR.
- Arm strongly recommends that you do not use R12 for R<sub>t</sub>.
- R<sub>t2</sub> must be R(t + 1).

### Use of PC

In A32 instructions you can use PC for R<sub>t</sub> in STR word instructions and PC for R<sub>n</sub> in STR instructions with immediate offset syntax (that is the forms that do not writeback to the R<sub>n</sub>). However, this is deprecated.

Other uses of PC are not permitted in these A32 instructions.

In T32 code, using PC in STR instructions is not permitted.

### Use of SP

You can use SP for R<sub>n</sub>.

In A32 code, you can use SP for R<sub>t</sub> in word instructions. You can use SP for R<sub>t</sub> in non-word instructions in A32 code but this is deprecated.

In T32 code, you can use SP for R<sub>t</sub> in word instructions only. All other use of SP for R<sub>t</sub> in this instruction is not permitted in T32 code.

### Example

```
STR      r2,[r9,#consta-struc] ; consta-struc is an expression
                                ; evaluating to a constant in
                                ; the range 0-4095.
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

<sup>z</sup> Must be divisible by 4.

<sup>aa</sup> R<sub>t</sub> and R<sub>n</sub> must be in the range R0-R7.

<sup>ab</sup> R<sub>t</sub> must be in the range R0-R7.

<sup>ac</sup> Must be divisible by 2.

## C2.143 STR (register offset)

Store with register offset, pre-indexed register offset, or post-indexed register offset.

### Syntax

```
STR{type}{cond} Rt, [Rn, ±Rm {, shift}] ; register offset
STR{type}{cond} Rt, [Rn, ±Rm {, shift}]! ; pre-indexed ; A32 only
STR{type}{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed ; A32 only
STRD{cond} Rt, Rt2, [Rn, ±Rm] ; register offset, doubleword ; A32 only
STRD{cond} Rt, Rt2, [Rn, ±Rm]! ; pre-indexed, doubleword ; A32 only
STRD{cond} Rt, Rt2, [Rn], ±Rm ; post-indexed, doubleword ; A32 only
```

where:

*type*

can be any one of:

**B**

Byte

**H**

Halfword

-

omitted, for Word.

*cond*

is an optional condition code.

*Rt*

is the general-purpose register to store.

*Rn*

is the general-purpose register on which the memory address is based.

*Rm*

is a general-purpose register containing a value to be used as the offset. *-Rm* is not permitted in T32 code.

*shift*

is an optional shift.

*Rt2*

is the additional register to store for doubleword operations.

Not all options are available in every instruction set and architecture.

### Offset register and shift options

The following table shows the ranges of offsets and availability of this instruction:

Table C2-16 Options and architectures, STR (register offsets)

Instruction	±Rm ad	shift		
A32, word or byte	±Rm	LSL #0-31	LSR #1-32	
		ASR #1-32	ROR #1-31	RRX
A32, halfword	±Rm	Not available		
A32, doubleword	±Rm	Not available		

<sup>ad</sup> Where ±Rm is shown, you can use -Rm, +Rm, or Rm. Where +Rm is shown, you cannot use -Rm.  
<sup>ae</sup> Rt, Rn, and Rm must all be in the range R0-R7.

**Table C2-16 Options and architectures, STR (register offsets) (continued)**

Instruction	$\pm Rm$ <small>ad</small>	shift		
T32 32-bit encoding, word, halfword, or byte	+Rm	LSL #0-3		
T32 16-bit encoding, all except doubleword <small>ae</small>	+Rm	Not available		

### Register restrictions

In the pre-index and post-index forms,  $Rn$  must be different from  $Rt$ .

#### Doubleword register restrictions

For A32 instructions:

- $Rt$  must be an even-numbered register.
- $Rt$  must not be LR.
- Arm strongly recommends that you do not use R12 for  $Rt$ .
- $Rt2$  must be  $R(t + 1)$ .
- $Rn$  must be different from  $Rt2$  in the pre-index and post-index forms.

### Use of PC

In A32 instructions you can use PC for  $Rt$  in STR word instructions, and you can use PC for  $Rn$  in STR instructions with register offset syntax (that is, the forms that do not writeback to the  $Rn$ ). However, this is deprecated.

Other uses of PC are not permitted in A32 instructions.

Use of PC in STR T32 instructions is not permitted.

### Use of SP

You can use SP for  $Rn$ .

In A32 code, you can use SP for  $Rt$  in word instructions. You can use SP for  $Rt$  in non-word A32 instructions but this is deprecated.

You can use SP for  $Rm$  in A32 instructions but this is deprecated.

In T32 code, you can use SP for  $Rt$  in word instructions only. All other use of SP for  $Rt$  in this instruction is not permitted in T32 code.

Use of SP for  $Rm$  is not permitted in T32 state.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.144 STR, unprivileged

Unprivileged Store, byte, halfword, or word.

### Syntax

STR{*type*}T{*cond*} *Rt*, [*Rn* {, #*offset*}] ; immediate offset (T32, 32-bit encoding only)

STR{*type*}T{*cond*} *Rt*, [*Rn*] {, #*offset*} ; post-indexed (A32 only)

STR{*type*}T{*cond*} *Rt*, [*Rn*], ±*Rm* {, *shift*} ; post-indexed (register) (A32 only)

where:

*type*

can be any one of:

**B**

Byte

**H**

Halfword

-

omitted, for Word.

*cond*

is an optional condition code.

*Rt*

is the register to load or store.

*Rn*

is the register on which the memory address is based.

*offset*

is an offset. If offset is omitted, the address is the value in *Rn*.

*Rm*

is a register containing a value to be used as the offset. *Rm* must not be PC.

*shift*

is an optional shift.

### Operation

When these instructions are executed by privileged software, they access memory with the same restrictions as they would have if they were executed by unprivileged software.

When executed by unprivileged software, these instructions behave in exactly the same way as the corresponding store instruction, for example STRBT behaves in the same way as STRB.

### Offset ranges and architectures

The following table shows the ranges of offsets and availability of this instruction:

**Table C2-17 Offsets and architectures, STR (User mode)**

Instruction	Immediate offset	Post-indexed	+/-Rm <sup>af</sup>	shift
A32, word or byte	Not available	-4095 to 4095	+/-Rm	LSL #0-31
				LSR #1-32
				ASR #1-32
				ROR #1-31
				RRX
A32, halfword	Not available	-255 to 255	+/-Rm	Not available
T32 32-bit encoding, word, halfword, or byte	0 to 255	Not available	Not available	Not available

**Related reference**

[C1.9 Condition code suffixes on page C1-142](#)

---

<sup>af</sup> You can use -Rm, +Rm, or Rm.

## C2.145 STREX

Store Register Exclusive.

### Syntax

STREX{cond} Rd, Rt, [Rn {, #offset}]

STREXB{cond} Rd, Rt, [Rn]

STREXH{cond} Rd, Rt, [Rn]

STREXD{cond} Rd, Rt, Rt2, [Rn]

where:

*cond*

is an optional condition code.

*Rd*

is the destination register for the returned status.

*Rt*

is the register to store.

*Rt2*

is the second register for doubleword stores.

*Rn*

is the register on which the memory address is based.

*offset*

is an optional offset applied to the value in *Rn*. *offset* is permitted only in T32 instructions. If *offset* is omitted, an offset of 0 is assumed.

### Operation

STREX performs a conditional store to memory. The conditions are as follows:

- If the physical address does not have the Shared TLB attribute, and the executing processor has an outstanding tagged physical address, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding tagged physical address, the store does not take place, and the value 1 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place, and the value 1 is returned in *Rd*.

### Restrictions

PC must not be used for any of *Rd*, *Rt*, *Rt2*, or *Rn*.

For STREX, *Rd* must not be the same register as *Rt*, *Rt2*, or *Rn*.

For A32 instructions:

- SP can be used but use of SP for any of *Rd*, *Rt*, or *Rt2* is deprecated.
- For STREXD, *Rt* must be an even numbered register, and not LR.

- $Rt2$  must be  $R(t+1)$ .
- *offset* is not permitted.

For T32 instructions:

- SP can be used for  $Rn$ , but must not be used for any of  $Rd$ ,  $Rt$ , or  $Rt2$ .
- The value of *offset* can be any multiple of four in the range 0-1020.

## Usage

Use LDREX and STREX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding LDREX and STREX instructions to a minimum.

### Note

The address used in a STREX instruction must be the same as the address in the most recently executed LDREX instruction.

## Availability

All these 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions.

## Examples

```
try      MOV r1, #0x1          ; load the 'lock taken' value
        LDREX r0, [LockAddr]    ; load the lock value
        CMP r0, #0              ; is the lock free?
        STREXEQ r0, r1, [LockAddr] ; try and claim the lock
        CMPEQ r0, #0            ; did this succeed?
        BNE try                ; no - try again
        ....                  ; yes - we have the lock
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.146 SUB

Subtract without carry.

### Syntax

```
SUB{S}{cond} {Rd}, Rn, Operand2  
SUB{cond} {Rd}, Rn, #imm12 ; T32, 32-bit encoding only  
where:
```

**S**

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Operand2*

is a flexible second operand.

*imm12*

is any value in the range 0-4095.

### Operation

The SUB instruction subtracts the value of *Operand2* or *imm12* from the value in *Rn*.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

### Use of PC and SP in T32 instructions

In general, you cannot use PC (R15) for *Rd*, or any operand. The exception is you can use PC for *Rn* in 32-bit T32 SUB instructions, with a constant *Operand2* value in the range 0-4095, and no S suffix. These instructions are useful for generating PC-relative addresses. Bit[1] of the PC value reads as 0 in this case, so that the base address for the calculation is always word-aligned.

Generally, you cannot use SP (R13) for *Rd*, or any operand, except that you can use SP for *Rn*.

### Use of PC and SP in A32 instructions

You cannot use PC for *Rd* or any operand in a SUB instruction that has a register-controlled shift.

In SUB instructions without register-controlled shift, use of PC is deprecated except for the following cases:

- Use of PC for *Rd*.
- Use of PC for *Rn* in the instruction SUB{cond} Rd, Rn, #Constant.

If you use PC (R15) as *Rn* or *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the SUBS pc,lr instruction.

You can use SP for *Rn* in SUB instructions, however, SUBS PC, SP, #Constant is deprecated.

You can use SP in SUB (register) if *Rn* is SP and *shift* is omitted or LSL #1, LSL #2, or LSL #3.

Other uses of SP in A32 SUB instructions are deprecated.

————— Note ————

Use of SP and PC is deprecated in A32 instructions.

### Condition flags

If S is specified, the SUB instruction updates the N, Z, C and V flags according to the result.

### 16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

**SUBS Rd, Rn, Rm**

*Rd*, *Rn* and *Rm* must all be Lo registers. This form can only be used outside an IT block.

**SUB{cond} Rd, Rn, Rm**

*Rd*, *Rn* and *Rm* must all be Lo registers. This form can only be used inside an IT block.

**SUBS Rd, Rn, #imm**

*imm* range 0-7. *Rd* and *Rn* must both be Lo registers. This form can only be used outside an IT block.

**SUB{cond} Rd, Rn, #imm**

*imm* range 0-7. *Rd* and *Rn* must both be Lo registers. This form can only be used inside an IT block.

**SUBS Rd, Rd, #imm**

*imm* range 0-255. *Rd* must be a Lo register. This form can only be used outside an IT block.

**SUB{cond} Rd, Rd, #imm**

*imm* range 0-255. *Rd* must be a Lo register. This form can only be used inside an IT block.

**SUB{cond} SP, SP, #imm**

*imm* range 0-508, word aligned.

### Example

```
SUBS    r8, r6, #240      ; sets the flags based on the result
```

### Multiword arithmetic examples

These instructions subtract one 96-bit integer contained in R9, R10, and R11 from another 96-bit integer contained in R6, R7, and R8, and place the result in R3, R4, and R5:

```
SUBS    r3, r6, r9
SBCS    r4, r7, r10
SBC     r5, r8, r11
```

For clarity, the above examples use consecutive registers for multiword values. There is no requirement to do this. The following, for example, is perfectly valid:

```
SUBS    r6, r6, r9
SBCS    r9, r2, r1
SBC     r2, r8, r11
```

**Related reference**

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.147 SUBS pc, lr on page C2-370](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.147 SUBS pc, lr

Exception return, without popping anything from the stack.

### Syntax

```
SUBS{cond} pc, lr, #imm ; A32 and T32 code  
MOVS{cond} pc, lr ; A32 and T32 code  
op1S{cond} pc, Rn, #imm ; A32 code only and is deprecated  
op1S{cond} pc, Rn, Rm {, shift} ; A32 code only and is deprecated  
op2S{cond} pc, #imm ; A32 code only and is deprecated  
op2S{cond} pc, Rm {, shift} ; A32 code only and is deprecated
```

where:

*op1*

is one of ADC, ADD, AND, BIC, EOR, ORN, ORR, RSB, RSC, SBC, and SUB.

*op2*

is one of MOV and MVN.

*cond*

is an optional condition code.

*imm*

is an immediate value. In T32 code, it is limited to the range 0-255. In A32 code, it is a flexible second operand.

*Rn*

is the first general-purpose source register. Arm deprecates the use of any register except LR.

*Rm*

is the optionally shifted second or only general-purpose register.

*shift*

is an optional condition code.

### Usage

SUBS pc, lr, #imm subtracts a value from the link register and loads the PC with the result, then copies the SPSR to the CPSR.

You can use SUBS pc, lr, #imm to return from an exception if there is no return state on the stack. The value of #imm depends on the exception to return from.

### Notes

SUBS pc, lr, #imm writes an address to the PC. The alignment of this address must be correct for the instruction set in use after the exception return:

- For a return to A32, the address written to the PC must be word-aligned.
- For a return to T32, the address written to the PC must be halfword-aligned.
- For a return to Jazelle, there are no alignment restrictions on the address written to the PC.

No special precautions are required in software to follow these rules, if you use the instruction to return after a valid exception entry mechanism.

In T32, only `SUBS{cond} pc, lr, #imm` is a valid instruction. `MOVS pc, lr` is a synonym of `SUBS pc, lr, #0`. Other instructions are undefined.

In A32, only `SUBS{cond} pc, lr, #imm` and `MOVS{cond} pc, lr` are valid instructions. Other instructions are deprecated.

———— Caution ————

Do not use these instructions in User mode or System mode. The assembler cannot warn you about this.

## Availability

This 32-bit instruction is available in A32 and T32.

The 32-bit T32 instruction is not available in the Armv7-M architecture.

There is no 16-bit version of this instruction in T32.

## Related reference

[C2.12 AND on page C2-178](#)

[C2.61 MOV on page C2-252](#)

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C2.9 ADD on page C2-171](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.148 SVC

SuperVisor Call.

### Syntax

`SVC{cond} #imm`

where:

*cond*

is an optional condition code.

*imm*

is an expression evaluating to an integer in the range:

- 0 to  $2^{24}-1$  (a 24-bit value) in an A32 instruction.
- 0-255 (an 8-bit value) in a T32 instruction.

### Operation

The SVC instruction causes an exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SVC vector.

*imm* is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

————— Note —————

SVC was called SWI in earlier versions of the A32 assembly language. SWI instructions disassemble to SVC, with a comment to say that this was formerly SWI.

### Condition flags

This instruction does not change the flags.

### Availability

This instruction is available in A32 and 16-bit T32 and in the Armv7 architectures.

There is no 32-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C2.149 SWP and SWPB

Swap data between registers and memory.

— Note —

These instruction are not supported in Armv8.

### Syntax

`SWP{B}{cond} Rt, Rt2, [Rn]`

where:

*cond*

is an optional condition code.

**B**

is an optional suffix. If **B** is present, a byte is swapped. Otherwise, a 32-bit word is swapped.

*Rt*

is the destination register. *Rt* must not be PC.

*Rt2*

is the source register. *Rt2* can be the same register as *Rt*. *Rt2* must not be PC.

*Rn*

contains the address in memory. *Rn* must be a different register from both *Rt* and *Rt2*. *Rn* must not be PC.

### Usage

You can use SWP and SWPB to implement semaphores:

- Data from memory is loaded into *Rt*.
- The contents of *Rt2* are saved to memory.
- If *Rt2* is the same register as *Rt*, the contents of the register are swapped with the contents of the memory location.

### Note

The use of SWP and SWPB is deprecated. You can use LDREX and STREX instructions to implement more sophisticated semaphores.

### Availability

These instructions are available in A32.

There are no T32 SWP or SWPB instructions.

### Related reference

[C2.54 LDREX on page C2-242](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.150 SXTAB

Sign extend Byte with Add, to extend an 8-bit value to a 32-bit value.

### Syntax

`SXTAB{cond} {Rd}, Rn, Rm {,rotation}`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the number to add.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

This instruction does the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits[7:0] from the value obtained.
3. Sign extend to 32 bits.
4. Add the value from *Rn*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C1.9 Condition code suffixes on page C1-142*

## C2.151 SXTAB16

Sign extend two Bytes with Add, to extend two 8-bit values to two 16-bit values.

### Syntax

`SXTAB16{cond} {Rd}, Rn, Rm {,rotation}`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the number to add.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

This instruction does the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits[23:16] and bits[7:0] from the value obtained.
3. Sign extend to 16 bits.
4. Add them to bits[31:16] and bits[15:0] respectively of *Rn* to form bits[31:16] and bits[15:0] of the result.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C1.9 Condition code suffixes* on page C1-142

## C2.152 SXTAH

Sign extend Halfword with Add, to extend a 16-bit value to a 32-bit value.

### Syntax

`SXTAH{cond} {Rd}, Rn, Rm {,rotation}`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the number to add.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

This instruction does the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits[15:0] from the value obtained.
3. Sign extend to 32 bits.
4. Add the value from *Rn*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C1.9 Condition code suffixes* on page C1-142

## C2.153 SXTB

Sign extend Byte, to extend an 8-bit value to a 32-bit value.

### Syntax

**SXTB{cond} {Rd}, Rm {,rotation}**

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

This instruction does the following:

1. Rotates the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracts bits[7:0] from the value obtained.
3. Sign extends to 32 bits.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

**SXTB Rd, Rm**

*Rd* and *Rm* must both be Lo registers.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

The 16-bit instruction is available in T32.

***Related reference***

[C1.9 Condition code suffixes on page C1-142](#)

## C2.154 SXTB16

Sign extend two bytes.

### Syntax

`SXTB16{cond} {Rd}, Rm [,rotation]`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

SXTB16 extends two 8-bit values to two 16-bit values. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[23:16] and bits[7:0] from the value obtained.
3. Sign extending to 16 bits each.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.155 SXT<sub>H</sub>

Sign extend Halfword.

### Syntax

SXT<sub>H</sub>{cond} {Rd}, Rm {,rotation}

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

SXT<sub>H</sub> extends a 16-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[15:0] from the value obtained.
3. Sign extending to 32 bits.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

**SXT<sub>H</sub> Rd, Rm**

*Rd* and *Rm* must both be Lo registers.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

The 16-bit instruction is available in T32.

### Example

```
SXTTH      r3, r9
```

### Incorrect example

```
SXTTH      r3, r9, ROR #12 ; rotation must be 0, 8, 16, or 24.
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.156 SYS

Execute system coprocessor instruction.

### Syntax

`SYS{cond} instruction{, Rn}`

where:

*cond*

is an optional condition code.

*instruction*

is the coprocessor instruction to execute.

*Rn*

is an operand to the instruction. For instructions that take an argument, *Rn* is compulsory. For instructions that do not take an argument, *Rn* is optional and if it is not specified, R0 is used. *Rn* must not be PC.

### Usage

You can use this pseudo-instruction to execute special coprocessor instructions such as cache, branch predictor, and TLB operations. The instructions operate by writing to special write-only coprocessor registers. The instruction names are the same as the write-only coprocessor register names and are listed in the *Arm® Architecture Reference Manual*. For example:

```
SYS ICIAALLUIS ; invalidates all instruction caches Inner Shareable
; to Point of Unification and also flushes branch
; target cache.
```

### Availability

This 32-bit instruction is available in A32 and T32.

The 32-bit T32 instruction is not available in the Armv7-M architecture.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

### Related information

[Arm Architecture Reference Manual](#)

## C2.157 TBB and TBH

Table Branch Byte and Table Branch Halfword.

### Syntax

TBB [*Rn*, *Rm*]  
TBH [*Rn*, *Rm*, LSL #1]

where:

*Rn*

is the base register. This contains the address of the table of branch lengths. *Rn* must not be SP.

If PC is specified for *Rn*, the value used is the address of the instruction plus 4.

*Rm*

is the index register. This contains an index into the table.

*Rm* must not be PC or SP.

### Operation

These instructions cause a PC-relative forward branch using a table of single byte offsets (TBB) or halfword offsets (TBH). *Rn* provides a pointer to the table, and *Rm* supplies an index into the table. The branch length is twice the value of the byte (TBB) or the halfword (TBH) returned from the table. The target of the branch table must be in the same execution state.

### Architectures

These 32-bit T32 instructions are available.

There are no versions of these instructions in A32 or in 16-bit T32 encodings.

## C2.158 TEQ

Test Equivalence.

### Syntax

TEQ{cond} *Rn*, *Operand2*

where:

*cond*

is an optional condition code.

*Rn*

is the general-purpose register holding the first operand.

*Operand2*

is a flexible second operand.

### Usage

This instruction tests the value in a register against *Operand2*. It updates the condition flags on the result, but does not place the result in any register.

The TEQ instruction performs a bitwise Exclusive OR operation on the value in *Rn* and the value of *Operand2*. This is the same as an EORS instruction, except that the result is discarded.

Use the TEQ instruction to test if two values are equal, without affecting the V or C flags (as CMP does).

TEQ is also useful for testing the sign of a value. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

### Register restrictions

In this T32 instruction, you cannot use SP or PC for *Rn* or *Operand2*.

In this A32 instruction, use of SP or PC is deprecated.

For A32 instructions:

- If you use PC (R15) as *Rn*, the value used is the address of the instruction plus 8.
- You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

### Condition flags

This instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

### Architectures

This instruction is available in A32 and T32.

### Correct example

```
TEQEQ    r10, r9
```

### Incorrect example

```
TEQ      pc, r1, ROR r0      ; PC not permitted with register
; controlled shift
```

**Related reference**

[C2.3 Flexible second operand \(Operand2\) on page C2-162](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.159 TST

Test bits.

### Syntax

TST{cond} *Rn*, *Operand2*

where:

*cond*

is an optional condition code.

*Rn*

is the general-purpose register holding the first operand.

*Operand2*

is a flexible second operand.

### Operation

This instruction tests the value in a register against *Operand2*. It updates the condition flags on the result, but does not place the result in any register.

The TST instruction performs a bitwise AND operation on the value in *Rn* and the value of *Operand2*. This is the same as an ANDS instruction, except that the result is discarded.

### Register restrictions

In this T32 instruction, you cannot use SP or PC for *Rn* or *Operand2*.

In this A32 instruction, use of SP or PC is deprecated.

For A32 instructions:

- If you use PC (R15) as *Rn*, the value used is the address of the instruction plus 8.
- You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

### Condition flags

This instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

### 16-bit instructions

The following form of the TST instruction is available in T32 code, and is a 16-bit instruction:

**TST *Rn*, *Rm***

*Rn* and *Rm* must both be Lo registers.

### Architectures

This instruction is available A32 and T32.

### Examples

```
TST      r0, #0x3F8
TSTNE   r1, r5, ASR r1
```

### Related reference

[C2.3 Flexible second operand \(\*Operand2\*\) on page C2-162](#)

*C1.9 Condition code suffixes on page C1-142*

## C2.160 TT, TTT, TTA, TTAT

Test Target (Alternate Domain, Unprivileged).

### Syntax

```
TT{cond}{q} Rd, Rn ; T1 TT general registers (T32)
TTA{cond}{q} Rd, Rn ; T1 TTA general registers (T32)
TTAT{cond}{q} Rd, Rn ; T1 TTAT general registers (T32)
TTT{cond}{q} Rd, Rn ; T1 TTT general registers (T32)
```

Where:

#### *cond*

Is an optional condition code. It specifies the condition under which the instruction is executed.  
If *cond* is omitted, it defaults to *always* (AL). See [Chapter C1 Condition Codes](#) on page [C1-133](#).

#### *q*

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page [C2-161](#).

#### *Rd*

Is the destination general-purpose register into which the status result of the target test is written.

#### *Rn*

Is the general-purpose base register.

### Usage

Test Target (TT) queries the security state and access permissions of a memory location.

Test Target Unprivileged (TTT) queries the security state and access permissions of a memory location for an unprivileged access to that location.

Test Target Alternate Domain (TTA) and Test Target Alternate Domain Unprivileged (TTAT) query the security state and access permissions of a memory location for a Non-secure access to that location.  
These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

These instructions return the security state and access permissions in the destination register, the contents of which are as follows:

Bits	Name	Description
[7:0]	MREGION	The MPU region that the address maps to. This field is 0 if MRVALID is 0.
[15:8]	SREGION	The SAU region that the address maps to. This field is only valid if the instruction is executed from Secure state. This field is 0 if SRVALID is 0.
[16]	MRVALID	Set to 1 if the MREGION content is valid. Set to 0 if the MREGION content is invalid.
[17]	SRVALID	Set to 1 if the SREGION content is valid. Set to 0 if the SREGION content is invalid.
[18]	R	Read accessibility. Set to 1 if the memory location can be read according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this bit returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.
[19]	RW	Read/write accessibility. Set to 1 if the memory location can be read and written according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this bit returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.
[20]	NSR	Equal to R AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU/IDAU permissions. This bit is only valid if the instruction is executed from Secure state and the R field is valid.

(continued)

Bits	Name	Description
[21]	NSRW	Equal to RW AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU/IDAU permissions. This bit is only valid if the instruction is executed from Secure state and the RW field is valid.
[22]	S	Security. A value of 1 indicates the memory location is Secure, and a value of 0 indicates the memory location is Non-secure. This bit is only valid if the instruction is executed from Secure state.
[23]	IRVALID	IREGION valid flag. For a Secure request, indicates the validity of the IREGION field. Set to 1 if the IREGION content is valid. Set to 0 if the IREGION content is invalid.  This bit is always 0 if the IDAU cannot provide a region number, the address is exempt from security attribution, or if the requesting TT instruction is executed from the Non-secure state.
[31:24]	IREGION	IDAU region number. Indicates the IDAU region number containing the target address. This field is 0 if IRVALID is 0.

Invalid fields are 0.

The MREGION field is invalid and 0 if any of the following conditions are true:

- The MPU is not present or MPU\_CTRL.ENABLE is 0.
- The address did not match any enabled MPU regions.
- The address matched multiple MPU regions.
- TT or TTT was executed from an unprivileged mode.

The SREGION field is invalid and 0 if any of the following conditions are true:

- SAU\_CTRL.ENABLE is set to 0.
- The address did not match any enabled SAU regions.
- The address matched multiple SAU regions.
- The SAU attributes were overridden by the IDAU.
- The instruction is executed from Non-secure state, or is executed on a processor that does not implement the Armv8-M Security Extensions.

The R and RW bits are invalid and 0 if any of the following conditions are true:

- The address matched multiple MPU regions.
- TT or TTT is executed from an unprivileged mode.

#### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

[C2.2 Instruction width specifiers](#) on page C2-161

## C2.161 UADD8

Unsigned parallel byte-wise addition.

### Syntax

`UADD8{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs four unsigned integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. The results are modulo 2<sup>8</sup>. It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[0]**

for bits[7:0] of the result.

**GE[1]**

for bits[15:8] of the result.

**GE[2]**

for bits[23:16] of the result.

**GE[3]**

for bits[31:24] of the result.

It sets a GE flag to 1 to indicate that the corresponding result overflowed, generating a carry. This is equivalent to an ADDS instruction setting the C condition flag to 1.

You can use these flags to control a following SEL instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C2.103 SEL* on page C2-310

*C1.9 Condition code suffixes* on page C1-142

## C2.162 UADD16

Unsigned parallel halfword-wise addition.

### Syntax

`UADD16{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs two unsigned integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. The results are modulo  $2^{16}$ . It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[1:0]**

for bits[15:0] of the result.

**GE[3:2]**

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result overflowed, generating a carry. This is equivalent to an ADDS instruction setting the C condition flag to 1.

You can use these flags to control a following SEL instruction.

---

#### Note

---

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

---

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C2.103 SEL on page C2-310*

*C1.9 Condition code suffixes on page C1-142*

## C2.163 UASX

Unsigned parallel add and subtract halfwords with exchange.

### Syntax

`UASX{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. The results are modulo  $2^{16}$ . It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[1:0]**

for bits[15:0] of the result.

**GE[3:2]**

for bits[31:16] of the result.

It sets GE[1:0] to 1 to indicate that the subtraction gave a result greater than or equal to zero, meaning a borrow did not occur. This is equivalent to a SUBS instruction setting the C condition flag to 1.

It sets GE[3:2] to 1 to indicate that the addition overflowed, generating a carry. This is equivalent to an ADDS instruction setting the C condition flag to 1.

You can use these flags to control a following SEL instruction.

---

#### Note

---

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

---

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

[C2.103 SEL on page C2-310](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.164 UBFX

Unsigned Bit Field Extract.

### Syntax

`UBFX{cond} Rd, Rn, #lsb, #width`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the source register.

*lsb*

is the bit number of the least significant bit in the bitfield, in the range 0 to 31.

*width*

is the width of the bitfield, in the range 1 to (32–*lsb*).

### Operation

Copies adjacent bits from one register into the least significant bits of a second register, and zero extends to 32 bits.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not alter any flags.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.165 UDF

Permanently Undefined.

### Syntax

`UDF{c}{q} {#}imm ; A1 general registers (A32)`

`UDF{c}{q} {#}imm ; T1 general registers (T32)`

`UDF{c}.W {#}imm ; T2 general registers (T32)`

Where:

*imm*

The value depends on the instruction variant:

#### A1 general registers

For A32, a 16-bit unsigned immediate, in the range 0 to 65535.

#### T1 general registers

For T32, an 8-bit unsigned immediate, in the range 0 to 255.

#### T2 general registers

For T32, a 16-bit unsigned immediate, in the range 0 to 65535.

— Note —

The PE ignores the value of this constant.

*c*

Is an optional condition code. See [Chapter C1 Condition Codes](#) on page C1-133. Arm deprecates using any *c* value other than AL.

*q*

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

### Usage

Permanently Undefined generates an Undefined Instruction exception.

The encodings for UDF used in this section are defined as permanently UNDEFINED in the Armv8-A architecture. However:

- With the T32 instruction set, Arm deprecates using the UDF instruction in an IT block.
- In the A32 instruction set, UDF is not conditional.

### Related reference

[C2.1 A32 and T32 instruction summary](#) on page C2-156

## C2.166 UDIV

Unsigned Divide.

### Syntax

UDIV{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the value to be divided.

*Rm*

is a register holding the divisor.

### Register restrictions

PC or SP cannot be used for Rd, Rn, or Rm.

### Architectures

This 32-bit T32 instruction is available in Armv7-R, Armv7-M and Armv8-M Mainline.

This 32-bit A32 instruction is optional in Armv7-R.

This 32-bit A32 and T32 instruction is available in Armv7-A if Virtualization Extensions are implemented, and optional if not.

There is no 16-bit T32 UDIV instruction.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.167 UHADD8

Unsigned halving parallel byte-wise addition.

### Syntax

`UHADD8{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs four unsigned integer additions on the corresponding bytes of the operands, halves the results, and writes the results into the corresponding bytes of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.168 UHADD16

Unsigned halving parallel halfword-wise addition.

### Syntax

`UHADD16{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs two unsigned integer additions on the corresponding halfwords of the operands, halves the results, and writes the results into the corresponding halfwords of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.169 UHASX

Unsigned halving parallel add and subtract halfwords with exchange.

### Syntax

`UHASX{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It halves the results and writes them into the corresponding halfwords of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.170 UHSAX

Unsigned halving parallel subtract and add halfwords with exchange.

### Syntax

`UHSAX{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It halves the results and writes them into the corresponding halfwords of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.171 UHSUB8

Unsigned halving parallel byte-wise subtraction.

### Syntax

`UHSUB8{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand, halves the results, and writes the results into the corresponding bytes of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.172 UHSUB16

Unsigned halving parallel halfword-wise subtraction.

### Syntax

`UHSUB16{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand, halves the results, and writes the results into the corresponding halfwords of the destination. This cannot cause overflow.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.173 UMAAL

Unsigned Multiply Accumulate Accumulate Long.

### Syntax

UMAAL{cond} RdLo, RdHi, Rn, Rm

where:

*cond*

is an optional condition code.

RdLo, RdHi

are the destination registers for the 64-bit result. They also hold the two 32-bit accumulate operands. RdLo and RdHi must be different registers.

Rn, Rm

are the general-purpose registers holding the multiply operands.

### Operation

The UMAAL instruction multiplies the 32-bit values in Rn and Rm, adds the two 32-bit values in RdHi and RdLo, and stores the 64-bit result to RdLo, RdHi.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Examples

UMAAL	r8, r9, r2, r3
UMAALGE	r2, r0, r5, r3

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.174 UMLAL

Unsigned Long Multiply, with optional Accumulate, with 32-bit operands and 64-bit result and accumulator.

### Syntax

`UMLAL{S}{cond} RdLo, RdHi, Rn, Rm`

where:

**S**

is an optional suffix available in A32 state only. If S is specified, the condition flags are updated based on the result of the operation.

*cond*

is an optional condition code.

*RdLo, RdHi*

are the destination registers. They also hold the accumulating value. *RdLo* and *RdHi* must be different registers.

*Rn, Rm*

are general-purpose registers holding the operands.

### Operation

The UMLAL instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies these integers, and adds the 64-bit result to the 64-bit unsigned integer contained in *RdHi* and *RdLo*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

If S is specified, this instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Example

```
UMLALS      r4, r5, r3, r8
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.175 UMULL

Unsigned Long Multiply, with 32-bit operands, and 64-bit result.

### Syntax

UMULL{S}{cond} RdLo, RdHi, Rn, Rm

where:

**S**

is an optional suffix available in A32 state only. If S is specified, the condition flags are updated based on the result of the operation.

*cond*

is an optional condition code.

RdLo, RdHi

are the destination general-purpose registers. RdLo and RdHi must be different registers.

Rn, Rm

are general-purpose registers holding the operands.

### Operation

The UMULL instruction interprets the values from Rn and Rm as unsigned integers. It multiplies these integers and places the least significant 32 bits of the result in RdLo, and the most significant 32 bits of the result in RdHi.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

If S is specified, this instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

### Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Example

```
UMULL      r0, r4, r5, r6
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.176 UQADD8

Unsigned saturating parallel byte-wise addition.

### Syntax

`UQADD8{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs four unsigned integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. It saturates the results to the unsigned range  $0 \leq x \leq 2^8 - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.177 UQADD16

Unsigned saturating parallel halfword-wise addition.

### Syntax

`UQADD16{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction performs two unsigned integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range  $0 \leq x \leq 2^{16} - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.178 UQASX

Unsigned saturating parallel add and subtract halfwords with exchange.

### Syntax

`UQASX{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range  $0 \leq x \leq 2^{16} - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.179 UQSAX

Unsigned saturating parallel subtract and add halfwords with exchange.

### Syntax

`UQSAX{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range  $0 \leq x \leq 2^{16}$  -1. The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.180 UQSUB8

Unsigned saturating parallel byte-wise subtraction.

### Syntax

`UQSUB8{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand and writes the results into the corresponding bytes of the destination. It saturates the results to the unsigned range  $0 \leq x \leq 2^8 - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.181 UQSUB16

Unsigned saturating parallel halfword-wise subtraction.

### Syntax

`UQSUB16{cond} {Rd}, Rn, Rm`

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range  $0 \leq x \leq 2^{16} - 1$ . The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.182 USAD8

Unsigned Sum of Absolute Differences.

### Syntax

USAD8{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Rm*

is the register holding the second operand.

### Operation

The USAD8 instruction finds the four differences between the unsigned values in corresponding bytes of *Rn* and *Rm*. It adds the absolute values of the four differences, and saves the result to *Rd*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not alter any flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

USAD8 r2, r4, r6

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.183 USADA8

Unsigned Sum of Absolute Differences and Accumulate.

### Syntax

USADA8{cond} Rd, Rn, Rm, Ra

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the first operand.

*Rm*

is the register holding the second operand.

*Ra*

is the register holding the accumulate operand.

### Operation

The USADA8 instruction adds the absolute values of the four differences to the value in *Ra*, and saves the result to *Rd*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not alter any flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Correct examples

```
USADA8      r0, r3, r5, r2
USADA8VS    r0, r4, r0, r1
```

### Incorrect examples

```
USADA8      r2, r4, r6      ; USADA8 requires four registers
USADA16    r0, r4, r0, r1  ; no such instruction
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.184 USAT

Unsigned Saturate to any bit position, with optional shift before saturating.

### Syntax

`USAT{cond} Rd, #sat, Rm{, shift}`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*sat*

specifies the bit position to saturate to, in the range 0 to 31.

*Rm*

is the register containing the operand.

*shift*

is an optional shift. It must be one of the following:

**ASR #n**

where *n* is in the range 1-32 (A32) or 1-31 (T32).

**LSL #n**

where *n* is in the range 0-31.

### Operation

The USAT instruction applies the specified shift to a signed value, then saturates to the unsigned range  $0 \leq x \leq 2^{\text{sat}} - 1$ .

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an MRS instruction.

### Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Example

```
USATNE r0, #7, r5
```

### Related reference

[C2.134 SSAT16](#) on page C2-345

[C2.65 MRS \(PSR to general-purpose register\)](#) on page C2-257

[C1.9 Condition code suffixes](#) on page C1-142

## C2.185 USAT16

Parallel halfword Saturate.

### Syntax

`USAT16{cond} Rd, #sat, Rn`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*sat*

specifies the bit position to saturate to, in the range 0 to 15.

*Rn*

is the register holding the operand.

### Operation

Halfword-wise unsigned saturation to any bit position.

The USAT16 instruction saturates each signed halfword to the unsigned range  $0 \leq x \leq 2^{\text{sat}} - 1$ .

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Q flag

If saturation occurs on either halfword, this instruction sets the Q flag. To read the state of the Q flag, use an MRS instruction.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

```
USAT16 r0, #7, r5
```

### Related reference

[C2.65 MRS \(PSR to general-purpose register\) on page C2-257](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.186 USAX

Unsigned parallel subtract and add halfwords with exchange.

### Syntax

USAX{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. The results are modulo  $2^{16}$ . It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[1:0]**

for bits[15:0] of the result.

**GE[3:2]**

for bits[31:16] of the result.

It sets GE[1:0] to 1 to indicate that the addition overflowed, generating a carry. This is equivalent to an ADDS instruction setting the C condition flag to 1.

It sets GE[3:2] to 1 to indicate that the subtraction gave a result greater than or equal to zero, meaning a borrow did not occur. This is equivalent to a SUBS instruction setting the C condition flag to 1.

You can use these flags to control a following SEL instruction.

---

#### Note

---

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C2.103 SEL* on page C2-310

*C1.9 Condition code suffixes* on page C1-142

## C2.187 USUB8

Unsigned parallel byte-wise subtraction.

### Syntax

USUB8{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand and writes the results into the corresponding bytes of the destination. The results are modulo 2<sup>8</sup>. It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[0]**

for bits[7:0] of the result.

**GE[1]**

for bits[15:8] of the result.

**GE[2]**

for bits[23:16] of the result.

**GE[3]**

for bits[31:24] of the result.

It sets a GE flag to 1 to indicate that the corresponding result is greater than or equal to zero, meaning a borrow did not occur. This is equivalent to a SUBS instruction setting the C condition flag to 1.

You can use these flags to control a following SEL instruction.

### Availability

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.103 SEL on page C2-310](#)

*C1.9 Condition code suffixes on page C1-142*

## C2.188 USUB16

Unsigned parallel halfword-wise subtraction.

### Syntax

USUB16{cond} {Rd}, Rn, Rm

where:

*cond*

is an optional condition code.

*Rd*

is the destination general-purpose register.

*Rm, Rn*

are the general-purpose registers holding the operands.

### Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. The results are modulo  $2^{16}$ . It sets the APSR GE flags.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

**GE[1:0]**

for bits[15:0] of the result.

**GE[3:2]**

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero, meaning a borrow did not occur. This is equivalent to a SUBS instruction setting the C condition flag to 1.

You can use these flags to control a following SEL instruction.

---

———— Note ————

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

---

### Availability

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

### Related reference

[C2.103 SEL on page C2-310](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C2.189 UXTAB

Zero extend Byte and Add.

### Syntax

`UXTAB{cond} {Rd}, Rn, Rm {,rotation}`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the number to add.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

UXTAB extends an 8-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[7:0] from the value obtained.
3. Zero extending to 32 bits.
4. Adding the value from *Rn*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C1.9 Condition code suffixes* on page C1-142

## C2.190 UXTAB16

Zero extend two Bytes and Add.

### Syntax

`UXTAB16{cond} {Rd}, Rn, Rm {,rotation}`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the number to add.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

UXTAB16 extends two 8-bit values to two 16-bit values. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[23:16] and bits[7:0] from the value obtained.
3. Zero extending them to 16 bits.
4. Adding them to bits[31:16] and bits[15:0] respectively of *Rn* to form bits[31:16] and bits[15:0] of the result.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Example

```
UXTAB16EQ    r0, r0, r4, ROR #16
```

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.191 UXTAH

Zero extend Halfword and Add.

### Syntax

UXTAH{cond} {Rd}, Rn, Rm {,rotation}

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rn*

is the register holding the number to add.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

UXTAH extends a 16-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[15:0] from the value obtained.
3. Zero extending to 32 bits.
4. Adding the value from *Rn*.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

***Related reference***

*C1.9 Condition code suffixes* on page C1-142

## C2.192 UXTB

Zero extend Byte.

### Syntax

UXTB{*cond*} {*Rd*}, *Rm* {,*rotation*}

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

UXTB extends an 8-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16, or 24 bits.
2. Extracting bits[7:0] from the value obtained.
3. Zero extending to 32 bits.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instruction

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

**UXTB Rd, Rm**

*Rd* and *Rm* must both be Lo registers.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

The 16-bit instruction is available in T32.

***Related reference***

[C1.9 Condition code suffixes on page C1-142](#)

## C2.193 UXBTB16

Zero extend two Bytes.

### Syntax

UXTB16{cond} {Rd}, Rm {,rotation}

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

UXTB16 extends two 8-bit values to two 16-bit values. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[23:16] and bits[7:0] from the value obtained.
3. Zero extending each to 16 bits.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C2.194 UXTH

Zero extend Halfword.

### Syntax

`UXTH{cond} {Rd}, Rm {,rotation}`

where:

*cond*

is an optional condition code.

*Rd*

is the destination register.

*Rm*

is the register holding the value to extend.

*rotation*

is one of:

**ROR #8**

Value from *Rm* is rotated right 8 bits.

**ROR #16**

Value from *Rm* is rotated right 16 bits.

**ROR #24**

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

### Operation

UXTH extends a 16-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16, or 24 bits.
2. Extracting bits[15:0] from the value obtained.
3. Zero extending to 32 bits.

### Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

**UXTH Rd, Rm**

*Rd* and *Rm* must both be Lo registers.

### Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

The 16-bit instruction is available in T32.

***Related reference***

[C1.9 Condition code suffixes on page C1-142](#)

## C2.195 WFE

Wait For Event.

### Syntax

`WFE{cond}`

where:

*cond*

is an optional condition code.

### Operation

This is a hint instruction. It is optional whether this instruction is implemented or not. If this instruction is not implemented, it executes as a NOP. The assembler produces a diagnostic message if the instruction executes as a NOP on the target.

If the Event Register is not set, WFE suspends execution until one of the following events occurs:

- An IRQ interrupt, unless masked by the CPSR I-bit.
- An FIQ interrupt, unless masked by the CPSR F-bit.
- An Imprecise Data abort, unless masked by the CPSR A-bit.
- A Debug Entry request, if Debug is enabled.
- An Event signaled by another processor using the SEV instruction, or by the current processor using the SEVL instruction.

If the Event Register is set, WFE clears it and returns immediately.

If WFE is implemented, SEV must also be implemented.

### Availability

This instruction is available in A32 and T32.

#### *Related reference*

[C2.71 NOP on page C2-266](#)

[C1.9 Condition code suffixes on page C1-142](#)

[C2.106 SEV on page C2-314](#)

[C2.107 SEVL on page C2-315](#)

[C2.196 WFI on page C2-438](#)

## C2.196 WFI

Wait for Interrupt.

### Syntax

`WFI{cond}`

where:

*cond*

is an optional condition code.

### Operation

This is a hint instruction. It is optional whether this instruction is implemented or not. If this instruction is not implemented, it executes as a NOP. The assembler produces a diagnostic message if the instruction executes as a NOP on the target.

`WFI` suspends execution until one of the following events occurs:

- An IRQ interrupt, regardless of the CPSR I-bit.
- An FIQ interrupt, regardless of the CPSR F-bit.
- An Imprecise Data abort, unless masked by the CPSR A-bit.
- A Debug Entry request, regardless of whether Debug is enabled.

### Availability

This instruction is available in A32 and T32.

#### Related reference

[C2.71 NOP on page C2-266](#)

[C1.9 Condition code suffixes on page C1-142](#)

[C2.195 WFE on page C2-437](#)

## C2.197 YIELD

Yield.

### Syntax

`YIELD{cond}`

where:

*cond*

is an optional condition code.

### Operation

This is a hint instruction. It is optional whether this instruction is implemented or not. If this instruction is not implemented, it executes as a NOP. The assembler produces a diagnostic message if the instruction executes as a NOP on the target.

YIELD indicates to the hardware that the current thread is performing a task, for example a spinlock, that can be swapped out. Hardware can use this hint to suspend and resume threads in a multithreading system.

### Availability

This instruction is available in A32 and T32.

#### *Related reference*

[C2.71 NOP on page C2-266](#)

[C1.9 Condition code suffixes on page C1-142](#)



# Chapter C3

## Advanced SIMD Instructions (32-bit)

Describes Advanced SIMD assembly language instructions.

It contains the following sections:

- [\*C3.1 Summary of Advanced SIMD instructions\* on page C3-445.](#)
- [\*C3.2 Summary of shared Advanced SIMD and floating-point instructions\* on page C3-448.](#)
- [\*C3.3 Interleaving provided by load and store element and structure instructions\* on page C3-449.](#)
- [\*C3.4 Alignment restrictions in load and store element and structure instructions\* on page C3-450.](#)
- [\*C3.5 FLDMDBX, FLDMIAX\* on page C3-451.](#)
- [\*C3.6 FSTMDBX, FSTMIAZ\* on page C3-452.](#)
- [\*C3.7 VABA and VABAL\* on page C3-453.](#)
- [\*C3.8 VABD and VABDL\* on page C3-454.](#)
- [\*C3.9 VABS\* on page C3-455.](#)
- [\*C3.10 VACLE, VACLT, VACGE and VACGT\* on page C3-456.](#)
- [\*C3.11 VADD\* on page C3-457.](#)
- [\*C3.12 VADDHN\* on page C3-458.](#)
- [\*C3.13 VADDL and VADDW\* on page C3-459.](#)
- [\*C3.14 VAND \(immediate\)\* on page C3-460.](#)
- [\*C3.15 VAND \(register\)\* on page C3-461.](#)
- [\*C3.16 VBIC \(immediate\)\* on page C3-462.](#)
- [\*C3.17 VBIC \(register\)\* on page C3-463.](#)
- [\*C3.18 VBIF\* on page C3-464.](#)
- [\*C3.19 VBIT\* on page C3-465.](#)
- [\*C3.20 VBSL\* on page C3-466.](#)
- [\*C3.21 VCADD\* on page C3-467.](#)
- [\*C3.22 VCEQ \(immediate #0\)\* on page C3-468.](#)
- [\*C3.23 VCEQ \(register\)\* on page C3-469.](#)

- C3.24 *VCGE* (*immediate #0*) on page C3-470.
- C3.25 *VCGE* (*register*) on page C3-471.
- C3.26 *VCGT* (*immediate #0*) on page C3-472.
- C3.27 *VCGT* (*register*) on page C3-473.
- C3.28 *VCLE* (*immediate #0*) on page C3-474.
- C3.29 *VCLS* on page C3-475.
- C3.30 *VCLE* (*register*) on page C3-476.
- C3.31 *VCLT* (*immediate #0*) on page C3-477.
- C3.32 *VCLT* (*register*) on page C3-478.
- C3.33 *VCLZ* on page C3-479.
- C3.34 *VCMLA* on page C3-480.
- C3.35 *VCMLA* (*by element*) on page C3-481.
- C3.36 *VCNT* on page C3-482.
- C3.37 *VCVT* (*between fixed-point or integer, and floating-point*) on page C3-483.
- C3.38 *VCVT* (*between half-precision and single-precision floating-point*) on page C3-484.
- C3.39 *VCVT* (*from floating-point to integer with directed rounding modes*) on page C3-485.
- C3.40 *VCVTB, VCVTT* (*between half-precision and double-precision*) on page C3-486.
- C3.41 *VDUP* on page C3-487.
- C3.42 *VEOR* on page C3-488.
- C3.43 *VEXT* on page C3-489.
- C3.44 *VFMA, VFMS* on page C3-490.
- C3.45 *VFMAL* (*by scalar*) on page C3-491.
- C3.46 *VFMAL* (*vector*) on page C3-492.
- C3.47 *VFMSL* (*by scalar*) on page C3-493.
- C3.48 *VFMSL* (*vector*) on page C3-494.
- C3.49 *VHADD* on page C3-495.
- C3.50 *VHSUB* on page C3-496.
- C3.51 *VLDn* (*single n-element structure to one lane*) on page C3-497.
- C3.52 *VLDn* (*single n-element structure to all lanes*) on page C3-499.
- C3.53 *VLDn* (*multiple n-element structures*) on page C3-501.
- C3.54 *VLDM* on page C3-503.
- C3.55 *VLDR* on page C3-504.
- C3.56 *VLDR* (*post-increment and pre-decrement*) on page C3-505.
- C3.57 *VLDR* *pseudo-instruction* on page C3-506.
- C3.58 *VMAX* and *VMIN* on page C3-507.
- C3.59 *VMAXNM, VMINNM* on page C3-508.
- C3.60 *VMLA* on page C3-509.
- C3.61 *VMLA* (*by scalar*) on page C3-510.
- C3.62 *VMLAL* (*by scalar*) on page C3-511.
- C3.63 *VMLAL* on page C3-512.
- C3.64 *VMLS* (*by scalar*) on page C3-513.
- C3.65 *VMLS* on page C3-514.
- C3.66 *VMLSL* on page C3-515.
- C3.67 *VMLSL* (*by scalar*) on page C3-516.
- C3.68 *VMOV* (*immediate*) on page C3-517.
- C3.69 *VMOV* (*register*) on page C3-518.
- C3.70 *VMOV* (*between two general-purpose registers and a 64-bit extension register*) on page C3-519.
- C3.71 *VMOV* (*between a general-purpose register and an Advanced SIMD scalar*) on page C3-520.
- C3.72 *VMOVL* on page C3-521.
- C3.73 *VMOVN* on page C3-522.
- C3.74 *VMOV2* on page C3-523.
- C3.75 *VMRS* on page C3-524.
- C3.76 *VMSR* on page C3-525.
- C3.77 *VMUL* on page C3-526.
- C3.78 *VMUL* (*by scalar*) on page C3-527.

- [C3.79 VMULL](#) on page C3-528.
- [C3.80 VMULL \(by scalar\)](#) on page C3-529.
- [C3.81 VMVN \(register\)](#) on page C3-530.
- [C3.82 VMVN \(immediate\)](#) on page C3-531.
- [C3.83 VNEG](#) on page C3-532.
- [C3.84 VORN \(register\)](#) on page C3-533.
- [C3.85 VORN \(immediate\)](#) on page C3-534.
- [C3.86 VORR \(register\)](#) on page C3-535.
- [C3.87 VORR \(immediate\)](#) on page C3-536.
- [C3.88 VPADAL](#) on page C3-537.
- [C3.89 VPADD](#) on page C3-538.
- [C3.90 VPADDL](#) on page C3-539.
- [C3.91 VPMAX and VPMIN](#) on page C3-540.
- [C3.92 VPOP](#) on page C3-541.
- [C3.93 VPUSH](#) on page C3-542.
- [C3.94 VQABS](#) on page C3-543.
- [C3.95 VQADD](#) on page C3-544.
- [C3.96 VQDMLAL and VQDMLSL \(by vector or by scalar\)](#) on page C3-545.
- [C3.97 VQDMULH \(by vector or by scalar\)](#) on page C3-546.
- [C3.98 VQDMULL \(by vector or by scalar\)](#) on page C3-547.
- [C3.99 VQMOVN and VQMOVUN](#) on page C3-548.
- [C3.100 VQNEG](#) on page C3-549.
- [C3.101 VQRDMULH \(by vector or by scalar\)](#) on page C3-550.
- [C3.102 VQRSHL \(by signed variable\)](#) on page C3-551.
- [C3.103 VQRSHRN and VQRSHRUN \(by immediate\)](#) on page C3-552.
- [C3.104 VQSHL \(by signed variable\)](#) on page C3-553.
- [C3.105 VQSHL and VQSHLU \(by immediate\)](#) on page C3-554.
- [C3.106 VQSHRN and VQSHRUN \(by immediate\)](#) on page C3-555.
- [C3.107 VQSUB](#) on page C3-556.
- [C3.108 VRADDHN](#) on page C3-557.
- [C3.109 VRECPE](#) on page C3-558.
- [C3.110 VRECPs](#) on page C3-559.
- [C3.111 VREV16, VREV32, and VREV64](#) on page C3-560.
- [C3.112 VRHADD](#) on page C3-561.
- [C3.113 VRSHL \(by signed variable\)](#) on page C3-562.
- [C3.114 VRSHR \(by immediate\)](#) on page C3-563.
- [C3.115 VRSHRN \(by immediate\)](#) on page C3-564.
- [C3.116 VRINT](#) on page C3-565.
- [C3.117 VRSQRT](#) on page C3-566.
- [C3.118 VRSQRTS](#) on page C3-567.
- [C3.119 VRSRA \(by immediate\)](#) on page C3-568.
- [C3.120 VRSUBHN](#) on page C3-569.
- [C3.121 VSDOT \(vector\)](#) on page C3-570.
- [C3.122 VSDOT \(by element\)](#) on page C3-571.
- [C3.123 VSHL \(by immediate\)](#) on page C3-572.
- [C3.124 VSHL \(by signed variable\)](#) on page C3-573.
- [C3.125 VSHLL \(by immediate\)](#) on page C3-574.
- [C3.126 VSHR \(by immediate\)](#) on page C3-575.
- [C3.127 VSHRN \(by immediate\)](#) on page C3-576.
- [C3.128 VSLI](#) on page C3-577.
- [C3.129 VSRA \(by immediate\)](#) on page C3-578.
- [C3.130 VSRI](#) on page C3-579.
- [C3.131 VSTM](#) on page C3-580.
- [C3.132 VSTn \(multiple n-element structures\)](#) on page C3-581.
- [C3.133 VSTn \(single n-element structure to one lane\)](#) on page C3-583.
- [C3.134 VSTR](#) on page C3-585.

- [C3.135 VSTR \(post-increment and pre-decrement\) on page C3-586.](#)
- [C3.136 VSUB on page C3-587.](#)
- [C3.137 VSUBHN on page C3-588.](#)
- [C3.138 VSUBL and VSUBW on page C3-589.](#)
- [C3.139 VSWP on page C3-590.](#)
- [C3.140 VTBL and VTBX on page C3-591.](#)
- [C3.141 VTRN on page C3-592.](#)
- [C3.142 VTST on page C3-593.](#)
- [C3.143 VUDOT \(vector\) on page C3-594.](#)
- [C3.144 VUDOT \(by element\) on page C3-595.](#)
- [C3.145 VUZP on page C3-596.](#)
- [C3.146 VZIP on page C3-597.](#)

## C3.1 Summary of Advanced SIMD instructions

Most Advanced SIMD instructions are not available in floating-point.

The following table shows a summary of Advanced SIMD instructions that are not available as floating-point instructions:

**Table C3-1 Summary of Advanced SIMD instructions**

Mnemonic	Brief description
FLDMDBX, FLDMIAX	FLDMX
FSTMDBX, FSTMIAx	FSTMX
VABA, VABD	Absolute difference and Accumulate, Absolute Difference
VABS	Absolute value
VACGE, VACGT	Absolute Compare Greater than or Equal, Greater Than
VACLE, VACLT	Absolute Compare Less than or Equal, Less Than (pseudo-instructions)
VADD	Add
VADDHN	Add, select High half
VAND	Bitwise AND
VAND	Bitwise AND (pseudo-instruction)
VBIC	Bitwise Bit Clear (register)
VBIC	Bitwise Bit Clear (immediate)
VBIF, VBIT, VBSL	Bitwise Insert if False, Insert if True, Select
VCADD	Vector Complex Add
VCEQ, VCLE, VCLT	Compare Equal, Less than or Equal, Compare Less Than
VCGE, VCGT	Compare Greater than or Equal, Greater Than
VCLE, VCLT	Compare Less than or Equal, Compare Less Than (pseudo-instruction)
VCLS, VCLZ, VCNT	Count Leading Sign bits, Count Leading Zeros, and Count set bits
VCMLA	Vector Complex Multiply Accumulate
VCMLA (by element)	Vector Complex Multiply Accumulate (by element)
VCVT	Convert fixed-point or integer to floating-point, floating-point to integer or fixed-point
VCVT	Convert floating-point to integer with directed rounding modes
VCVT	Convert between half-precision and single-precision floating-point numbers
VDUP	Duplicate scalar to all lanes of vector
VEOR	Bitwise Exclusive OR
VEXT	Extract
VFMA, VFMS	Fused Multiply Accumulate, Fused Multiply Subtract
VFMAL, VFMSL	Vector Floating-point Multiply-Add Long to accumulator (by scalar)
VFMAL, VFMSL	Vector Floating-point Multiply-Add Long to accumulator (vector)
VHADD, VHSUB	Halving Add, Halving Subtract

**Table C3-1 Summary of Advanced SIMD instructions (continued)**

Mnemonic	Brief description
VLD	Vector Load
VMAX, VMIN	Maximum, Minimum
VMAXNM, VMINNM	Maximum, Minimum, consistent with IEEE 754-2008
VMLA, VMLS	Multiply Accumulate, Multiply Subtract (vector)
VMLA, VMLS	Multiply Accumulate, Multiply Subtract (by scalar)
VMOV	Move (immediate)
VMOV	Move (register)
VMOVL, VMOV{U}N	Move Long, Move Narrow (register)
VMUL	Multiply (vector)
VMUL	Multiply (by scalar)
VMVN	Move Negative (immediate)
VNEG	Negate
VORN	Bitwise OR NOT
VORN	Bitwise OR NOT (pseudo-instruction)
VORR	Bitwise OR (register)
VORR	Bitwise OR (immediate)
VPADD, VPADAL	Pairwise Add, Pairwise Add and Accumulate
VPMAX, VPMIN	Pairwise Maximum, Pairwise Minimum
VQABS	Absolute value, saturate
VQADD	Add, saturate
VQDMLAL, VQDMLSL	Saturating Doubling Multiply Accumulate, and Multiply Subtract
VQDMULL	Saturating Doubling Multiply
VQDMULH	Saturating Doubling Multiply returning High half
VQMOV{U}N	Saturating Move (register)
VQNEG	Negate, saturate
VQRDMULH	Saturating Doubling Multiply returning High half
VQRSHL	Shift Left, Round, saturate (by signed variable)
VQRSHR{U}N	Shift Right, Round, saturate (by immediate)
VQSHL	Shift Left, saturate (by immediate)
VQSHL	Shift Left, saturate (by signed variable)
VQSHR{U}N	Shift Right, saturate (by immediate)
VQSUB	Subtract, saturate
VRADDHN	Add, select High half, Round
VRECPE	Reciprocal Estimate

**Table C3-1 Summary of Advanced SIMD instructions (continued)**

Mnemonic	Brief description
VRECPS	Reciprocal Step
VREV	Reverse elements
VRHADD	Halving Add, Round
VRINT	Round to integer
VRSHR	Shift Right and Round (by immediate)
VRSHRN	Shift Right, Round, Narrow (by immediate)
VRSQRT	Reciprocal Square Root Estimate
VRSQRTS	Reciprocal Square Root Step
VRSRA	Shift Right, Round, and Accumulate (by immediate)
VRSUBHN	Subtract, select High half, Round
VSDOT (vector)	Dot Product vector form with signed integers
VSDOT (by element)	Dot Product index form with signed integers
VSHL	Shift Left (by immediate)
VSHR	Shift Right (by immediate)
VSHRN	Shift Right, Narrow (by immediate)
VSLI	Shift Left and Insert
VSRA	Shift Right, Accumulate (by immediate)
VSRI	Shift Right and Insert
VST	Vector Store
VSUB	Subtract
VSUBHN	Subtract, select High half
VSWP	Swap vectors
VTBL, VTBX	Vector table look-up
VTRN	Vector transpose
VTST	Test bits
VUDOT (vector)	Dot Product vector form with unsigned integers
VUDOT (by element)	Dot Product index form with unsigned integers
VUZP, VZIP	Vector interleave and de-interleave
VZIP	Vector Zip

## C3.2 Summary of shared Advanced SIMD and floating-point instructions

Some instructions are common to Advanced SIMD and floating-point.

The following table shows a summary of instructions that are common to the Advanced SIMD and floating-point instruction sets.

**Table C3-2 Summary of shared Advanced SIMD and floating-point instructions**

Mnemonic	Brief description
VLDM	Load multiple
VLDR	Load
	Load (post-increment and pre-decrement)
VMOV	Transfer from one general-purpose register to a scalar
	Transfer from two general-purpose registers to either one double-precision or two single-precision registers
	Transfer from a scalar to a general-purpose register
	Transfer from either one double-precision or two single-precision registers to two general-purpose registers
VMRS	Transfer from a SIMD and floating-point system register to a general-purpose register
VMSR	Transfer from a general-purpose register to a SIMD and floating-point system register
VPOP	Pop floating-point or SIMD registers from full-descending stack
VPUSH	Push floating-point or SIMD registers to full-descending stack
VSTM	Store multiple
VSTR	Store
	Store (post-increment and pre-decrement)

### Related reference

[C3.54 VLDM on page C3-503](#)

[C3.55 VLDR on page C3-504](#)

[C3.56 VLDR \(post-increment and pre-decrement\) on page C3-505](#)

[C3.57 VLDR pseudo-instruction on page C3-506](#)

[C3.70 VMOV \(between two general-purpose registers and a 64-bit extension register\) on page C3-519](#)

[C3.71 VMOV \(between a general-purpose register and an Advanced SIMD scalar\) on page C3-520](#)

[C3.75 VMRS on page C3-524](#)

[C3.76 VMSR on page C3-525](#)

[C3.92 VPOP on page C3-541](#)

[C3.93 VPUSH on page C3-542](#)

[C3.131 VSTM on page C3-580](#)

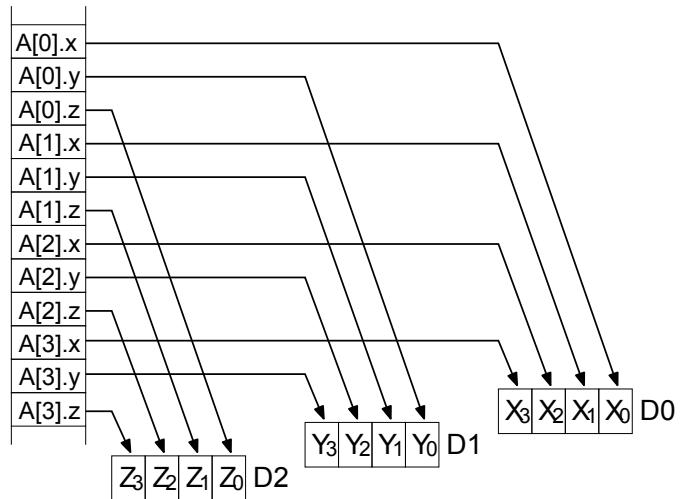
[C3.134 VSTR on page C3-585](#)

[C3.135 VSTR \(post-increment and pre-decrement\) on page C3-586](#)

### C3.3 Interleaving provided by load and store element and structure instructions

Many instructions in this group provide interleaving when structures are stored to memory, and de-interleaving when structures are loaded from memory.

The following figure shows an example of de-interleaving. Interleaving is the inverse process.



**Figure C3-1 De-interleaving an array of 3-element structures**

#### Related concepts

[C3.4 Alignment restrictions in load and store element and structure instructions](#) on page C3-450

#### Related reference

[C3.51 VLDn \(single n-element structure to one lane\)](#) on page C3-497

[C3.52 VLDn \(single n-element structure to all lanes\)](#) on page C3-499

[C3.53 VLDn \(multiple n-element structures\)](#) on page C3-501

[C3.132 VSTn \(multiple n-element structures\)](#) on page C3-581

[C3.133 VSTn \(single n-element structure to one lane\)](#) on page C3-583

#### Related information

[Arm Architecture Reference Manual](#)

## C3.4 Alignment restrictions in load and store element and structure instructions

Many of these instructions allow you to specify memory alignment restrictions.

When the alignment is not specified in the instruction, the alignment restriction is controlled by the A bit (SCTLR bit[1]):

- If the A bit is 0, there are no alignment restrictions (except for strongly-ordered or device memory, where accesses must be element-aligned).
- If the A bit is 1, accesses must be element-aligned.

If an address is not correctly aligned, an alignment fault occurs.

### **Related concepts**

[C3.3 Interleaving provided by load and store element and structure instructions](#) on page C3-449

### **Related reference**

[C3.51 VLDn \(single n-element structure to one lane\)](#) on page C3-497

[C3.52 VLDn \(single n-element structure to all lanes\)](#) on page C3-499

[C3.53 VLDn \(multiple n-element structures\)](#) on page C3-501

[C3.132 VSTn \(multiple n-element structures\)](#) on page C3-581

[C3.133 VSTn \(single n-element structure to one lane\)](#) on page C3-583

### **Related information**

[Arm Architecture Reference Manual](#)

## C3.5 FLDMDBX, FLDMIAX

FLDMX.

### Syntax

FLDMDBX{c}{q} Rn!, dreglist ; A1 Decrement Before FP/SIMD registers (A32)

FLDMIAX{c}{q} Rn{!}, dreglist ; A1 Increment After FP/SIMD registers (A32)

FLDMDBX{c}{q} Rn!, dreglist ; T1 Decrement Before FP/SIMD registers (T32)

FLDMIAX{c}{q} Rn{!}, dreglist ; T1 Increment After FP/SIMD registers (T32)

Where:

**c**

Is an optional condition code. See [Chapter C1 Condition Codes](#) on page C1-133.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**Rn**

Is the general-purpose base register. If writeback is not specified, the PC can be used.

**!**

Specifies base register writeback.

**dreglist**

Is the list of consecutively numbered 64-bit SIMD and FP registers to be transferred. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

### Usage

FLDMX loads multiple SIMD and FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

#### Note

---

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page C3-445

## C3.6 FSTMDBX, FSTMIAX

FSTMX.

### Syntax

FSTMDBX{c}{q} Rn!, dreglist ; A1 Decrement Before FP/SIMD registers (A32)

FSTMIAX{c}{q} Rn{!}, dreglist ; A1 Increment After FP/SIMD registers (A32)

FSTMDBX{c}{q} Rn!, dreglist ; T1 Decrement Before FP/SIMD registers (T32)

FSTMIAX{c}{q} Rn{!}, dreglist ; T1 Increment After FP/SIMD registers (T32)

Where:

**c**

Is an optional condition code. See [Chapter C1 Condition Codes](#) on page C1-133.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**Rn**

Is the general-purpose base register. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

**!**

Specifies base register writeback.

**dreglist**

Is the list of consecutively numbered 64-bit SIMD and FP registers to be transferred. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

### Usage

FSTMX stores multiple SIMD and FP registers from the Advanced SIMD and floating-point register file to consecutive locations in using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Note

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page C3-445

## C3.7 VABA and VABAL

Vector Absolute Difference and Accumulate.

### Syntax

VABA{cond}.datatype {Qd}, Qn, Qm

VABA{cond}.datatype {Dd}, Dn, Dm

VABAL{cond}.datatype Qd, Dn, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

*Qd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

### Operation

VABA subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

VABAL is the long version of the VABA instruction.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.8 VABD and VABDL

Vector Absolute Difference.

### Syntax

VABD{cond}.datatype {Qd}, Qn, Qm

VABD{cond}.datatype {Dd}, Dn, Dm

VABDL{cond}.datatype Qd, Dn, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of:

- S8, S16, S32, U8, U16, or U32 for VABDL.
- S8, S16, S32, U8, U16, U32 or F32 for VABD.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

*Qd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

### Operation

VABD subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results into the elements of the destination vector.

VABDL is the long version of the VABD instruction.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.9 VABS

Vector Absolute

### Syntax

`VABS{cond}.datatype Qd, Qm`

`VABS{cond}.datatype Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, or F32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VABS takes the absolute value of each element in a vector, and places the results in a second vector. (The floating-point version only clears the sign bit.)

#### Related reference

[C3.94 VQABS on page C3-543](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.10 VACLE, VACLT, VACGE and VACGT

Vector Absolute Compare.

### Syntax

`VACOp{cond}.F32 {Qd}, Qn, Qm`

`VACOp{cond}.F32 {Dd}, Dn, Dm`

where:

*op*

must be one of:

**GE**

Absolute Greater than or Equal.

**GT**

Absolute Greater Than.

**LE**

Absolute Less than or Equal.

**LT**

Absolute Less Than.

*cond*

is an optional condition code.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

The result datatype is I32.

### Operation

These instructions take the absolute value of each element in a vector, and compare it with the absolute value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

---

#### Note

On disassembly, the VACLE and VACLT pseudo-instructions are disassembled to the corresponding VACGE and VACGT instructions, with the operands reversed.

---

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.11 VADD

Vector Add.

### Syntax

VADD{cond}.datatype {Qd}, Qn, Qm

VADD{cond}.datatype {Dd}, Dn, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, I64, or F32

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VADD adds corresponding elements in two vectors, and places the results in the destination vector.

#### Related reference

[C3.13 VADDL and VADDW on page C3-459](#)

[C3.95 VQADD on page C3-544](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.12 VADDHN

Vector Add and Narrow, selecting High half.

### Syntax

`VADDHN{cond}.datatype Dd, Qn, Qm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or I64.

*Dd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector.

### Operation

VADDHN adds corresponding elements in two vectors, selects the most significant halves of the results, and places the final results in the destination vector. Results are truncated.

#### Related reference

[C3.108 VRADDHN on page C3-557](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.13 VADDL and VADDW

Vector Add Long, Vector Add Wide.

### Syntax

VADDL{cond}.datatype Qd, Dn, Dm ; Long operation

VADDW{cond}.datatype {Qd,} Qn, Dm ; Wide operation

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

*Qd, Qn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a wide operation.

### Operation

VADDL adds corresponding elements in two doubleword vectors, and places the results in the destination quadword vector.

VADDW adds corresponding elements in one quadword and one doubleword vector, and places the results in the destination quadword vector.

### Related reference

[C3.11 VADD on page C3-457](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.14 VAND (immediate)

Vector bitwise AND immediate pseudo-instruction.

### Syntax

`VAND{cond}.datatype Qd, #imm`

`VAND{cond}.datatype Dd, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be either I8, I16, I32, or I64.

*Qd* or *Dd*

is the Advanced SIMD register for the result.

*i*.**mm**

is the immediate value.

### Operation

VAND takes each element of the destination vector, performs a bitwise AND with an immediate value, and returns the result into the destination vector.

---

#### Note

---

On disassembly, this pseudo-instruction is disassembled to a corresponding VBIC instruction, with the complementary immediate value.

---

### Immediate values

If *datatype* is I16, the immediate value must have one of the following forms:

- 0xFFXY.
- 0xXYFF.

If *datatype* is I32, the immediate value must have one of the following forms:

- 0xFFFFFFFXY.
- 0xFFFFXXYYFF.
- 0xFFXYFFFF.
- 0xXYFFFFFF.

### Related reference

[C3.16 VBIC \(immediate\) on page C3-462](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.15 VAND (register)

Vector bitwise AND.

### Syntax

`VAND{cond}{.datatype} {Qd}, Qn, Qm`

`VAND{cond}{.datatype} {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional data type. The assembler ignores *datatype*.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VAND performs a bitwise logical AND between two registers, and places the result in the destination register.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.16 VBIC (immediate)

Vector Bit Clear immediate.

### Syntax

`VBIC{cond}.datatype Qd, #imm`

`VBIC{cond}.datatype Dd, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be either I8, I16, I32, or I64.

*Qd* or *Dd*

is the Advanced SIMD register for the source and result.

*imm*

is the immediate value.

### Operation

VBIC takes each element of the destination vector, performs a bitwise AND complement with an immediate value, and returns the result in the destination vector.

### Immediate values

You can either specify *imm* as a pattern which the assembler repeats to fill the destination register, or you can directly specify the immediate value (that conforms to the pattern) in full. The pattern for *imm* depends on *datatype* as shown in the following table:

Table C3-3 Patterns for immediate value in VBIC (immediate)

I16	I32
0x00XY	0x000000XY
0xXY00	0x0000XY00
	0x00XY0000
	0XXY000000

If you use the I8 or I64 datatypes, the assembler converts it to either the I16 or I32 instruction to match the pattern of *imm*. If the immediate value does not match any of the patterns in the preceding table, the assembler generates an error.

### Related reference

[C3.14 VAND \(immediate\) on page C3-460](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.17 VBIC (register)

Vector Bit Clear.

### Syntax

`VBIC{cond}{.datatype} {Qd}, Qn, Qm`

`VBIC{cond}{.datatype} {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional data type. The assembler ignores *datatype*.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VBIC performs a bitwise logical AND complement between two registers, and places the result in the destination register.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.18 VBIF

Vector Bitwise Insert if False.

### Syntax

`VBIF{cond}{.datatype} {Qd}, Qn, Qm`

`VBIF{cond}{.datatype} {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional datatype. The assembler ignores *datatype*.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VBIF inserts each bit from the first operand into the destination if the corresponding bit of the second operand is 0, otherwise it leaves the destination bit unchanged.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.19 VBIT

Vector Bitwise Insert if True.

### Syntax

`VBIT{cond}{.datatype} {Qd}, Qn, Qm`

`VBIT{cond}{.datatype} {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional datatype. The assembler ignores *datatype*.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

`VBIT` inserts each bit from the first operand into the destination if the corresponding bit of the second operand is 1, otherwise it leaves the destination bit unchanged.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.20 VBSL

Vector Bitwise Select.

### Syntax

`VBSL{cond}{.datatype} {Qd}, Qn, Qm`

`VBSL{cond}{.datatype} {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional datatype. The assembler ignores *datatype*.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VBSL selects each bit for the destination from the first operand if the corresponding bit of the destination is 1, or from the second operand if the corresponding bit of the destination is 0.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.21 VCADD

Vector Complex Add.

### Syntax

`VCADD{q}.dt {Dd,} Dn, Dm, #rotate ; A1 64-bit SIMD vector FP/SIMD registers (A32)`

`VCADD{q}.dt {Qd,} Qn, Qm, #rotate ; A1 128-bit SIMD vector FP/SIMD registers (A32)`

Where:

**Dd**

Is the 64-bit name of the SIMD and FP destination register.

**Dn**

Is the 64-bit name of the first SIMD and FP source register.

**Dm**

Is the 64-bit name of the second SIMD and FP source register.

**Qd**

Is the 128-bit name of the SIMD and FP destination register.

**Qn**

Is the 128-bit name of the first SIMD and FP source register.

**Qm**

Is the 128-bit name of the second SIMD and FP source register.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**dt**

Is the data type for the elements of the vectors, and can be either F16 or F32.

**rotate**

Is the rotation to be applied to elements in the second SIMD and FP source register, and can be either 90 or 270.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page C3-445

## C3.22 VCEQ (immediate #0)

Vector Compare Equal to zero.

### Syntax

`VCEQ{cond}.datatype {Qd}, Qn, #0`

`VCEQ{cond}.datatype {Dd}, Dn, #0`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, or F32.

The result datatype is:

- I32 for operand datatypes I32 or F32.
- I16 for operand datatype I16.
- I8 for operand datatype I8.

*Qd, Qn, Qm*

specifies the destination register and the operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register and the operand register, for a doubleword operation.

*#0*

specifies a comparison with zero.

### Operation

VCEQ takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.23 VCEQ (register)

Vector Compare Equal.

### Syntax

`VCEQ{cond}.datatype {Qd}, Qn, Qm`

`VCEQ{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, or F32.

The result datatype is:

- I32 for operand datatypes I32 or F32.
- I16 for operand datatype I16.
- I8 for operand datatype I8.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VCEQ takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.24 VCGE (immediate #0)

Vector Compare Greater than or Equal to zero.

### Syntax

`VCGE{cond}.datatype {Qd}, Qn, #0`

`VCGE{cond}.datatype {Dd}, Dn, #0`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, or F32.

The result datatype is:

- I32 for operand datatypes S32 or F32.
- I16 for operand datatype S16.
- I8 for operand datatype S8.

*Qd, Qn, Qm*

specifies the destination register and the operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register and the operand register, for a doubleword operation.

*#0*

specifies a comparison with zero.

### Operation

VCGE takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.25 VCGE (register)

Vector Compare Greater than or Equal.

### Syntax

`VCGE{cond}.datatype {Qd}, Qn, Qm`

`VCGE{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, U32, or F32.

The result datatype is:

- I32 for operand datatypes S32, U32, or F32.
- I16 for operand datatypes S16 or U16.
- I8 for operand datatypes S8 or U8.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VCGE takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.26 VCGT (immediate #0)

Vector Compare Greater Than zero.

### Syntax

`VCGT{cond}.datatype {Qd}, Qn, #0`

`VCGT{cond}.datatype {Dd}, Dn, #0`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, or F32.

The result datatype is:

- I32 for operand datatypes S32 or F32.
- I16 for operand datatype S16.
- I8 for operand datatype S8.

*Qd, Qn, Qm*

specifies the destination register and the operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register and the operand register, for a doubleword operation.

### Operation

VCGT takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.27 VCGT (register)

Vector Compare Greater Than.

### Syntax

`VCGT{cond}.datatype {Qd}, Qn, Qm`

`VCGT{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, U32, or F32.

The result datatype is:

- I32 for operand datatypes S32, U32, or F32.
- I16 for operand datatypes S16 or U16.
- I8 for operand datatypes S8 or U8.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VCGT takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.28 VCLE (immediate #0)

Vector Compare Less than or Equal to zero.

### Syntax

`VCLE{cond}.datatype {Qd}, Qn, #0`

`VCLE{cond}.datatype {Dd}, Dn, #0`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, or F32.

The result datatype is:

- I32 for operand datatypes S32 or F32.
- I16 for operand datatype S16.
- I8 for operand datatype S8.

*Qd, Qn, Qm*

specifies the destination register and the operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register and the operand register, for a doubleword operation.

*#0*

specifies a comparison with zero.

### Operation

VCLE takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.29 VCLS

Vector Count Leading Sign bits.

### Syntax

`VCLS{cond}.datatype Qd, Qm`

`VCLS{cond}.datatype Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, or S32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VCLS counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.30 VCLE (register)

Vector Compare Less than or Equal pseudo-instruction.

### Syntax

`VCLE{cond}.datatype {Qd}, Qn, Qm`

`VCLE{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, U32, or F32.

The result datatype is:

- I32 for operand datatypes S32, U32, or F32.
- I16 for operand datatypes S16 or U16.
- I8 for operand datatypes S8 or U8.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VCLE takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

On disassembly, this pseudo-instruction is disassembled to the corresponding VCGE instruction, with the operands reversed.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C3.31 VCLT (immediate #0)

Vector Compare Less Than zero.

### Syntax

`VCLT{cond}.datatype {Qd}, Qn, #0`

`VCLT{cond}.datatype {Dd}, Dn, #0`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, or F32.

The result datatype is:

- I32 for operand datatypes S32 or F32.
- I16 for operand datatype S16.
- I8 for operand datatype S8.

*Qd, Qn, Qm*

specifies the destination register and the operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register and the operand register, for a doubleword operation.

*#0*

specifies a comparison with zero.

### Operation

VCLT takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.32 VCLT (register)

Vector Compare Less Than.

### Syntax

`VCLT{cond}.datatype {Qd}, Qn, Qm`

`VCLT{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, U32, or F32.

The result datatype is:

- I32 for operand datatypes S32, U32, or F32.
- I16 for operand datatypes S16 or U16.
- I8 for operand datatypes S8 or U8.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VCLT takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

#### Note

On disassembly, this pseudo-instruction is disassembled to the corresponding VCGT instruction, with the operands reversed.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.33 VCLZ

Vector Count Leading Zeros.

### Syntax

`VCLZ{cond}.datatype Qd, Qm`

`VCLZ{cond}.datatype Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, or I32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VCLZ counts the number of consecutive zeros, starting from the top bit, in each element in a vector, and places the results in a second vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.34 VCMLA

Vector Complex Multiply Accumulate.

### Syntax

`VCMLA{q}.dt {Dd,} Dn, Dm, #rotate ; 64-bit SIMD vector FP/SIMD registers`

`VCMLA{q}.dt {Qd,} Qn, Qm, #rotate ; 128-bit SIMD vector FP/SIMD registers`

Where:

**Dd**

Is the 64-bit name of the SIMD and FP destination register.

**Dn**

Is the 64-bit name of the first SIMD and FP source register.

**Dm**

Is the 64-bit name of the second SIMD and FP source register.

**Qd**

Is the 128-bit name of the SIMD and FP destination register.

**Qn**

Is the 128-bit name of the first SIMD and FP source register.

**Qm**

Is the 128-bit name of the second SIMD and FP source register.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**dt**

Is the data type for the elements of the vectors, and can be either F16 or F32.

**rotate**

Is the rotation to be applied to elements in the second SIMD and FP source register, and can be one of 0, 90, 180 or 270.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page C3-445

## C3.35 VCMLA (by element)

Vector Complex Multiply Accumulate (by element).

### Syntax

`VCMLA{q}.F16 Dd, Dn, Dm[index], #rotate ; A1 Double,halfprec FP/SIMD registers (A32)`

`VCMLA{q}.F32 Dd, Dn, Dm[0], #rotate ; A1 Double,singleprec FP/SIMD registers (A32)`

`VCMLA{q}.F32 Qd, Qn, Dm[0], #rotate ; A1 Quad,singleprec FP/SIMD registers (A32)`

`VCMLA{q}.F16 Qd, Qn, Dm[index], #rotate ; A1 Halfprec,quad FP/SIMD registers (A32)`

Where:

**Dd**

Is the 64-bit name of the SIMD and FP destination register.

**Dn**

Is the 64-bit name of the first SIMD and FP source register.

**Dm**

Is the 64-bit name of the second SIMD and FP source register.

**index**

Is the element index in the range 0 to 1.

**Qd**

Is the 128-bit name of the SIMD and FP destination register.

**Qn**

Is the 128-bit name of the first SIMD and FP source register.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**rotate**

Is the rotation to be applied to elements in the second SIMD and FP source register, and can be one of 0, 90, 180 or 270.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page C3-445

## C3.36 VCNT

Vector Count set bits.

### Syntax

`VCNT{cond}.datatype Qd, Qm`

`VCNT{cond}.datatype Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be `I8`.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VCNT counts the number of bits that are one in each element in a vector, and places the results in a second vector.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.37 VCVT (between fixed-point or integer, and floating-point)

Vector Convert.

### Syntax

`VCVT{cond}.type Qd, Qm {, #fbits}`

`VCVT{cond}.type Dd, Dm {, #fbits}`

where:

*cond*

is an optional condition code.

*type*

specifies the data types for the elements of the vectors. It must be one of:

`S32.F32`

Floating-point to signed integer or fixed-point.

`U32.F32`

Floating-point to unsigned integer or fixed-point.

`F32.S32`

Signed integer or fixed-point to floating-point.

`F32.U32`

Unsigned integer or fixed-point to floating-point.

*Qd, Qm*

specifies the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

specifies the destination vector and the operand vector, for a doubleword operation.

*fbits*

if present, specifies the number of fraction bits in the fixed point number. Otherwise, the conversion is between floating-point and integer. *fbits* must lie in the range 0-32. If *fbits* is omitted, the number of fraction bits is 0.

### Operation

VCVT converts each element in a vector in one of the following ways, and places the results in the destination vector:

- From floating-point to integer.
- From integer to floating-point.
- From floating-point to fixed-point.
- From fixed-point to floating-point.

### Rounding

Integer or fixed-point to floating-point conversions use round to nearest.

Floating-point to integer or fixed-point conversions use round towards zero.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.38 VCVT (between half-precision and single-precision floating-point)

Vector Convert.

### Syntax

`VCVT{cond}.F32.F16 Qd, Dm`

`VCVT{cond}.F16.F32 Dd, Qm`

where:

*cond*

is an optional condition code.

*Qd, Dm*

specifies the destination vector for the single-precision results and the half-precision operand vector.

*Dd, Qm*

specifies the destination vector for half-precision results and the single-precision operand vector.

### Operation

VCVT with half-precision extension, converts each element in a vector in one of the following ways, and places the results in the destination vector:

- From half-precision floating-point to single-precision floating-point (F32.F16).
- From single-precision floating-point to half-precision floating-point (F16.F32).

### Architectures

This instruction is available in Armv8. In earlier architectures, it is only available in NEON systems with the half-precision extension.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C3.39 VCVT (from floating-point to integer with directed rounding modes)

VCVT (Vector Convert) converts each element in a vector from floating-point to signed or unsigned integer, and places the results in the destination vector.

————— Note —————

- This instruction is supported only in Armv8.
- You cannot use VCVT with a directed rounding mode inside an IT block.

### Syntax

`VCVTmode.type Qd, Qm`

`VCVTmode.type Dd, Dm`

where:

*mode*

must be one of:

**A**

meaning round to nearest, ties away from zero

**N**

meaning round to nearest, ties to even

**P**

meaning round towards plus infinity

**M**

meaning round towards minus infinity.

*type*

specifies the data types for the elements of the vectors. It must be one of:

**S32.F32**

floating-point to signed integer

**U32.F32**

floating-point to unsigned integer.

*Qd, Qm*

specifies the destination and operand vectors, for a quadword operation.

*Dd, Dm*

specifies the destination and operand vectors, for a doubleword operation.

## C3.40 VCVTB, VCVTT (between half-precision and double-precision)

These instructions convert between half-precision and double-precision floating-point numbers.

The conversion can be done in either of the following ways:

- From half-precision floating-point to double-precision floating-point (F64.F16).
- From double-precision floating-point to half-precision floating-point (F16.F64).

VCVTB uses the bottom half (bits[15:0]) of the single word register to obtain or store the half-precision value.

VCVTT uses the top half (bits[31:16]) of the single word register to obtain or store the half-precision value.

————— **Note** —————

These instructions are supported only in Armv8.

### Syntax

`VCVTB{cond}.F64.F16 Dd, Sm`

`VCVTB{cond}.F16.F64 Sd, Dm`

`VCVTT{cond}.F64.F16 Dd, Sm`

`VCVTT{cond}.F16.F64 Sd, Dm`

where:

*cond*

is an optional condition code.

*Dd*

is a double-precision register for the result.

*Sm*

is a single word register holding the operand.

*Sd*

is a single word register for the result.

*Dm*

is a double-precision register holding the operand.

### Usage

These instructions convert the half-precision value in *Sm* to double-precision and place the result in *Dd*, or the double-precision value in *Dm* to half-precision and place the result in *Sd*.

### Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

## C3.41 VDUP

Vector Duplicate.

### Syntax

`VDUP{cond}.size Qd, Dm[x]`

`VDUP{cond}.size Dd, Dm[x]`

`VDUP{cond}.size Qd, Rm`

`VDUP{cond}.size Dd, Rm`

where:

*cond*

is an optional condition code.

*size*

must be 8, 16, or 32.

*Qd*

specifies the destination register for a quadword operation.

*Dd*

specifies the destination register for a doubleword operation.

*Dm[x]*

specifies the Advanced SIMD scalar.

*Rm*

specifies the general-purpose register. *Rm* must not be PC.

### Operation

VDUP duplicates a scalar into every element of the destination vector. The source can be an Advanced SIMD scalar or a general-purpose register.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.42 VEOR

Vector Bitwise Exclusive OR.

### Syntax

`VEOR{cond}{.datatype} {Qd}, Qn, Qm`

`VEOR{cond}{.datatype} {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional data type. The assembler ignores *datatype*.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VEOR performs a logical exclusive OR between two registers, and places the result in the destination register.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.43 VEXT

Vector Extract.

### Syntax

`VEXT{cond}.8 {Qd}, Qn, Qm, #imm`

`VEXT{cond}.8 {Dd}, Dn, Dm, #imm`

where:

*cond*

is an optional condition code.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

*imm*

is the number of 8-bit elements to extract from the bottom of the second operand vector, in the range 0-7 for doubleword operations, or 0-15 for quadword operations.

### Operation

VEXT extracts 8-bit elements from the bottom end of the second operand vector and the top end of the first, concatenates them, and places the result in the destination vector. See the following figure for an example:

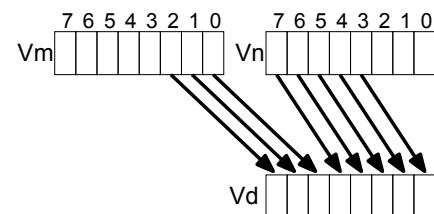


Figure C3-2 Operation of doubleword VEXT for imm = 3

### VEXT pseudo-instruction

You can specify a datatype of 16, 32, or 64 instead of 8. In this case, #imm refers to halfwords, words, or doublewords instead of referring to bytes, and the permitted ranges are correspondingly reduced.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.44 VFMA, VFMS

Vector Fused Multiply Accumulate, Vector Fused Multiply Subtract.

### Syntax

$vop\{cond\}.F32 \{Qd\}, Qn, Qm$

$vop\{cond\}.F32 \{Dd\}, Dn, Dm$

where:

$op$

is one of FMA or FMS.

$cond$

is an optional condition code.

$Dd, Dn, Dm$

are the destination and operand vectors for doubleword operation.

$Qd, Qn, Qm$

are the destination and operand vectors for quadword operation.

### Operation

VFMA multiplies corresponding elements in the two operand vectors, and accumulates the results into the elements of the destination vector. The result of the multiply is not rounded before the accumulation.

VFMS multiplies corresponding elements in the two operand vectors, then subtracts the products from the corresponding elements of the destination vector, and places the final results in the destination vector. The result of the multiply is not rounded before the subtraction.

### Related reference

[C3.77 VMUL on page C3-526](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.45 VFMA (by scalar)

Vector Floating-point Multiply-Add Long to accumulator (by scalar).

### Syntax

`VFMA{q}.F16 Dd, Sn, Sm[index] ; 64-bit SIMD vector`

`VFMA{q}.F16 Qd, Dn, Dm[index] ; 128-bit SIMD vector FP/SIMD registers (A32)`

Where:

**Dd**

Is the 64-bit name of the SIMD and FP destination register.

**Sn**

Is the 32-bit name of the first SIMD and FP source register.

**Sm**

Is the 32-bit name of the second SIMD and FP source register.

**index**

Depends on the instruction variant:

**64**

For the 64-bit SIMD vector variant: is the element index in the range 0 to 1.

**128**

For the 128-bit SIMD vector variant: is the element index in the range 0 to 3.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers on page C2-161](#).

**Qd**

Is the 128-bit name of the SIMD and FP destination register.

**Dn**

Is the 64-bit name of the first SIMD and FP source register.

**Dm**

Is the 64-bit name of the second SIMD and FP source register.

### Architectures supported

Supported in Armv8.2 and later.

### Usage

Vector Floating-point Multiply-Add Long to accumulator (by scalar). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the product to the corresponding vector element of the destination SIMD and FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPT, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

---

#### Note

---

ID\_ISAR6.FHM indicates whether this instruction is supported.

---

### Related reference

[C3.1 Summary of Advanced SIMD instructions on page C3-445](#)

## C3.46 VFML (vector)

Vector Floating-point Multiply-Add Long to accumulator (vector).

### Syntax

`VFML{q}.F16 Dd, Sn, Sm ; 64-bit SIMD vector`

`VFML{q}.F16 Qd, Dn, Dm ; 128-bit SIMD vector FP/SIMD registers (A32)`

Where:

**Dd**

Is the 64-bit name of the SIMD and FP destination register.

**Sn**

Is the 32-bit name of the first SIMD and FP source register.

**Sm**

Is the 32-bit name of the second SIMD and FP source register.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page [C2-161](#).

**Qd**

Is the 128-bit name of the SIMD and FP destination register.

**Dn**

Is the 64-bit name of the first SIMD and FP source register.

**Dm**

Is the 64-bit name of the second SIMD and FP source register.

### Architectures supported

Supported in Armv8.2 and later.

### Usage

Vector Floating-point Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding values in the vectors in the two source SIMD and FP registers, and accumulates the product to the corresponding vector element of the destination SIMD and FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

————— Note —————

ID\_ISAR6.FHM indicates whether this instruction is supported.

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page [C3-445](#)

## C3.47 VFMSL (by scalar)

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar).

### Syntax

```
VFMSL{q}.F16 Dd, Sn, Sm[index] ; 64-bit SIMD vector  
VFMSL{q}.F16 Qd, Dn, Dm[index] ; 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

**Dd**

Is the 64-bit name of the SIMD and FP destination register.

**Sn**

Is the 32-bit name of the first SIMD and FP source register.

**Sm**

Is the 32-bit name of the second SIMD and FP source register.

**index**

Depends on the instruction variant:

**64**

For the 64-bit SIMD vector variant: is the element index in the range 0 to 1.

**128**

For the 128-bit SIMD vector variant: is the element index in the range 0 to 3.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers on page C2-161](#).

**Qd**

Is the 128-bit name of the SIMD and FP destination register.

**Dn**

Is the 64-bit name of the first SIMD and FP source register.

**Dm**

Is the 64-bit name of the second SIMD and FP source register.

### Architectures supported

Supported in Armv8.2 and later.

### Usage

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar). This instruction multiplies the negated vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the product to the corresponding vector element of the destination SIMD and FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPT, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

---

**Note**

---

ID\_ISAR6.FHM indicates whether this instruction is supported.

---

### Related reference

[C3.1 Summary of Advanced SIMD instructions on page C3-445](#)

## C3.48 VFMSL (vector)

Vector Floating-point Multiply-Subtract Long from accumulator (vector).

### Syntax

`VFMSL{q}.F16 Dd, Sn, Sm ; 64-bit SIMD vector`

`VFMSL{q}.F16 Qd, Dn, Dm ; 128-bit SIMD vector FP/SIMD registers (A32)`

Where:

**Dd**

Is the 64-bit name of the SIMD and FP destination register.

**Sn**

Is the 32-bit name of the first SIMD and FP source register.

**Sm**

Is the 32-bit name of the second SIMD and FP source register.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page [C2-161](#).

**Qd**

Is the 128-bit name of the SIMD and FP destination register.

**Dn**

Is the 64-bit name of the first SIMD and FP source register.

**Dm**

Is the 64-bit name of the second SIMD and FP source register.

### Architectures supported

Supported in Armv8.2 and later.

### Usage

Vector Floating-point Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD and FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD and FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

---

#### Note

---

ID\_ISAR6.FHM indicates whether this instruction is supported.

---

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page [C3-445](#)

## C3.49 VHADD

Vector Halving Add.

### Syntax

`VHADD{cond}.datatype {Qd}, Qn, Qm`

`VHADD{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VHADD adds corresponding elements in two vectors, shifts each result right one bit, and places the results in the destination vector. Results are truncated.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.50 VHSUB

Vector Halving Subtract.

### Syntax

`VHSUB{cond}.datatype {Qd}, Qn, Qm`

`VHSUB{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VHSUB subtracts the elements of one vector from the corresponding elements of another vector, shifts each result right one bit, and places the results in the destination vector. Results are always truncated.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.51 VLDn (single n-element structure to one lane)

Vector Load single  $n$ -element structure to one lane.

### Syntax

`VLDn{cond}.datatype List, [Rn{@align}]{!}`

`VLDn{cond}.datatype List, [Rn{@align}], Rm`

where:

*n*

must be one of 1, 2, 3, or 4.

*cond*

is an optional condition code.

*datatype*

see the following table.

*List*

is the list of Advanced SIMD registers enclosed in braces, { and }. See the following table for options.

*Rn*

is the general-purpose register containing the base address. *Rn* cannot be PC.

*align*

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads have taken place.

*Rm*

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

### Operation

`VLDn` loads one  $n$ -element structure from memory into one or more Advanced SIMD registers. Elements of the register that are not loaded are unaltered.

**Table C3-4 Permitted combinations of parameters for VLDn (single n-element structure to one lane)**

<i>n</i>	<i>datatype</i>	<i>list ag</i>	<i>align ah</i>	<b>alignment</b>
1	8	{Dd[x]}	-	Standard only
	16	{Dd[x]}	@16	2-byte
	32	{Dd[x]}	@32	4-byte
2	8	{Dd[x], D(d+1)[x]}	@16	2-byte

**ag** Every register in the list must be in the range D0-D31.  
**ah** *align* can be omitted. In this case, standard alignment rules apply.

**Table C3-4 Permitted combinations of parameters for VLDn (single n-element structure to one lane) (continued)**

<i>n</i>	<i>datatype</i>	<i>list ag</i>	<i>align ah</i>	<i>alignment</i>
	16	{Dd[x], D(d+1)[x]}	@32	4-byte
		{Dd[x], D(d+2)[x]}	@32	4-byte
	32	{Dd[x], D(d+1)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x]}	@64	8-byte
3	8	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
	16 or 32	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
		{Dd[x], D(d+2)[x], D(d+4)[x]}	-	Standard only
4	8	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@32	4-byte
	16	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64	8-byte
	32	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64 or @128	8-byte or 16-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64 or @128	8-byte or 16-byte

**Related concepts**

[C3.3 Interleaving provided by load and store element and structure instructions](#) on page C3-449

**Related reference**

[C1.9 Condition code suffixes](#) on page C1-142

## C3.52 VLDn (single n-element structure to all lanes)

Vector Load single  $n$ -element structure to all lanes.

### Syntax

`VLDn{cond}.datatype List, [Rn{@align}]{!}`

`VLDn{cond}.datatype List, [Rn{@align}], Rm`

where:

*n*

must be one of 1, 2, 3, or 4.

*cond*

is an optional condition code.

*datatype*

see the following table.

*List*

is the list of Advanced SIMD registers enclosed in braces, { and }. See the following table for options.

*Rn*

is the general-purpose register containing the base address. *Rn* cannot be PC.

*align*

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads have taken place.

*Rm*

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

### Operation

`VLDn` loads multiple copies of one  $n$ -element structure from memory into one or more Advanced SIMD registers.

**Table C3-5 Permitted combinations of parameters for VLDn (single n-element structure to all lanes)**

<i>n</i>	<i>datatype</i>	<i>list</i> <sup>ai</sup>	<i>align</i> <sup>aj</sup>	<b>alignment</b>
1	8	{Dd[]}	-	Standard only
		{Dd[], D(d+1)[]}		Standard only
16		{Dd[]}	@16	2-byte
		{Dd[], D(d+1)[]}	@16	2-byte

<sup>ai</sup> Every register in the list must be in the range D0-D31.  
<sup>aj</sup> *align* can be omitted. In this case, standard alignment rules apply.

**Table C3-5 Permitted combinations of parameters for VLDn (single n-element structure to all lanes) (continued)**

<i>n</i>	<i>datatype</i>	<i>list ai</i>	<i>align aj</i>	<b>alignment</b>
32		{Dd[]}	@32	4-byte
		{Dd[], D(d+1)[]}	@32	4-byte
2	8	{Dd[], D(d+1)[]}	@8	byte
		{Dd[], D(d+2)[]}	@8	byte
2	16	{Dd[], D(d+1)[]}	@16	2-byte
		{Dd[], D(d+2)[]}	@16	2-byte
2	32	{Dd[], D(d+1)[]}	@32	4-byte
		{Dd[], D(d+2)[]}	@32	4-byte
3	8, 16, or 32	{Dd[], D(d+1)[], D(d+2)[]}	-	Standard only
		{Dd[], D(d+2)[], D(d+4)[]}	-	Standard only
4	8	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@32	4-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@32	4-byte
4	16	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@64	8-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@64	8-byte
4	32	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@64 or @128	8-byte or 16-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@64 or @128	8-byte or 16-byte

**Related concepts**

[C3.3 Interleaving provided by load and store element and structure instructions](#) on page C3-449

**Related reference**

[C1.9 Condition code suffixes](#) on page C1-142

## C3.53 VLDn (multiple n-element structures)

Vector Load multiple  $n$ -element structures.

### Syntax

`VLDn{cond}.datatype List, [Rn{@align}]{!}`  
`VLDn{cond}.datatype List, [Rn{@align}], Rm`

where:

*n*

must be one of 1, 2, 3, or 4.

*cond*

is an optional condition code.

*datatype*

see the following table for options.

*List*

is the list of Advanced SIMD registers enclosed in braces, { and }. See the following table for options.

*Rn*

is the general-purpose register containing the base address. *Rn* cannot be PC.

*align*

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads have taken place.

*Rm*

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

### Operation

`VLDn` loads multiple  $n$ -element structures from memory into one or more Advanced SIMD registers, with de-interleaving (unless  $n == 1$ ). Every element of each register is loaded.

**Table C3-6 Permitted combinations of parameters for VLDn (multiple n-element structures)**

<i>n</i>	<i>datatype</i>	<i>list ak</i>	<i>align al</i>	<i>alignment</i>
1	8, 16, 32, or 64	{Dd}	@64	8-byte
		{Dd, D(d+1)}	@64 or @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

*ak* Every register in the list must be in the range D0-D31.

*al* *align* can be omitted. In this case, standard alignment rules apply.

**Table C3-6 Permitted combinations of parameters for VLDn (multiple n-element structures) (continued)**

<i>n</i>	<i>datatype</i>	<i>list ak</i>	<i>align al</i>	<i>alignment</i>
2	8, 16, or 32	{Dd, D(d+1)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+2)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
3	8, 16, or 32	{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+2), D(d+4)}	@64	8-byte
4	8, 16, or 32	{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
		{Dd, D(d+2), D(d+4), D(d+6)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

**Related concepts**

[C3.3 Interleaving provided by load and store element and structure instructions](#) on page C3-449

**Related reference**

[C1.9 Condition code suffixes](#) on page C1-142

## C3.54 VLDM

Extension register load multiple.

### Syntax

`VLDMmode{cond} Rn{!}, Registers`

where:

*mode*

must be one of:

**IA**

meaning Increment address After each transfer. IA is the default, and can be omitted.

**DB**

meaning Decrement address Before each transfer.

**EA**

meaning Empty Ascending stack operation. This is the same as DB for loads.

**FD**

meaning Full Descending stack operation. This is the same as IA for loads.

*cond*

is an optional condition code.

*Rn*

is the general-purpose register holding the base address for the transfer.

!

is optional. ! specifies that the updated base address must be written back to *Rn*. If ! is not specified, *mode* must be IA.

*Registers*

is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify D or Q registers, but they must not be mixed. The number of registers must not exceed 16 D registers, or 8 Q registers. If Q registers are specified, on disassembly they are shown as D registers.

---

#### Note

---

VPOP *Registers* is equivalent to VLDM sp!, *Registers*.

You can use either form of this instruction. They both disassemble to VPOP.

---

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

[C4.14 VLDM \(floating-point\)](#) on page C4-615

## C3.55 VLDR

Extension register load.

### Syntax

`VLDR{cond}{.64} Dd, [Rn{, #offset}]`

`VLDR{cond}{.64} Dd, Label`

where:

*cond*

is an optional condition code.

*Dd*

is the extension register to be loaded.

*Rn*

is the general-purpose register holding the base address for the transfer.

*offset*

is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range -1020 to +1020. The value is added to the base address to form the address used for the transfer.

*Label*

is a PC-relative expression.

*Label* must be aligned on a word boundary within  $\pm 1\text{KB}$  of the current instruction.

### Operation

The VLDR instruction loads an extension register from memory.

Two words are transferred.

There is also a VLDR pseudo-instruction.

### Related reference

[C3.57 VLDR pseudo-instruction on page C3-506](#)

[C1.9 Condition code suffixes on page C1-142](#)

[C4.15 VLDR \(floating-point\) on page C4-616](#)

## C3.56 VLDR (post-increment and pre-decrement)

Pseudo-instruction that loads extension registers, with post-increment and pre-decrement forms.

————— Note —————

There are also VLDR and VSTR instructions without post-increment and pre-decrement.

### Syntax

VLDR{cond}{.64} Dd, [Rn], #offset ; post-increment

VLDR{cond}{.64} Dd, [Rn, #-offset]! ; pre-decrement

where:

*cond*

is an optional condition code.

*Dd*

is the extension register to load.

*Rn*

is the general-purpose register holding the base address for the transfer.

*offset*

is a numeric expression that must evaluate to 8 at assembly time.

### Operation

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. This pseudo-instruction assembles to a VLDM instruction.

### Related reference

[C3.54 VLDM on page C3-503](#)

[C3.55 VLDR on page C3-504](#)

[C1.9 Condition code suffixes on page C1-142](#)

[C4.16 VLDR \(post-increment and pre-decrement, floating-point\) on page C4-617](#)

## C3.57 VLDR pseudo-instruction

The VLDR pseudo-instruction loads a constant value into every element of a 64-bit Advanced SIMD vector.

— Note —

This description is for the VLDR pseudo-instruction only.

### Syntax

`VLDR{cond}.datatype Dd,=constant`

where:

*cond*

is an optional condition code.

*datatype*

must be one of *In*, *Sn*, *Un*, or *F32*.

*n*

must be one of 8, 16, 32, or 64.

*Dd*

is the extension register to be loaded.

*constant*

is an immediate value of the appropriate type for *datatype*.

### Usage

If an instruction (for example, *VMOV*) is available that can generate the constant directly into the register, the assembler uses it. Otherwise, it generates a doubleword literal pool entry containing the constant and loads the constant using a VLDR instruction.

#### Related reference

[C3.55 VLDR on page C3-504](#)

[C1.9 Condition code suffixes on page C1-142](#)

[C3.57 VLDR pseudo-instruction on page C3-506](#)

## C3.58 VMAX and VMIN

Vector Maximum, Vector Minimum.

### Syntax

$Vop\{cond\}.datatype\ Qd, Qn, Qm$

$Vop\{cond\}.datatype\ Dd, Dn, Dm$

where:

$op$

must be either MAX or MIN.

$cond$

is an optional condition code.

$datatype$

must be one of S8, S16, S32, U8, U16, U32, or F32.

$Qd, Qn, Qm$

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

$Dd, Dn, Dm$

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VMAX compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

VMIN compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

### Floating-point maximum and minimum

$\max(+0.0, -0.0) = +0.0$ .

$\min(+0.0, -0.0) = -0.0$

If any input is a NaN, the corresponding result element is the default NaN.

### Related reference

[C3.89 VPADD on page C3-538](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.59 VMAXNM, VMINNM

Vector Minimum, Vector Maximum.

————— Note ————

- These instructions are supported only in Armv8.
- You cannot use VMAXNM or VMINNM inside an IT block.

### Syntax

*Vop.F32 Qd, Qn, Qm*

*Vop.F32 Dd, Dn, Dm*

where:

*op*

must be either MAXNM or MINNM.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VMAXNM compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

VMINNM compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

If one of the elements in a pair is a number and the other element is NaN, the corresponding result element is the number. This is consistent with the IEEE 754-2008 standard.

## C3.60 VMLA

Vector Multiply Accumulate.

### Syntax

VMLA{cond}.datatype {Qd}, Qn, Qm

VMLA{cond}.datatype {Dd}, Dn, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, or F32.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VMLA multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.61 VMLA (by scalar)

Vector Multiply by scalar and Accumulate.

### Syntax

VMLA{cond}.datatype {Qd}, Qn, Dm[x]

VMLA{cond}.datatype {Dd}, Dn, Dm[x]

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or F32.

*Qd, Qn*

are the destination vector and the first operand vector, for a quadword operation.

*Dd, Dn*

are the destination vector and the first operand vector, for a doubleword operation.

*Dm[x]*

is the scalar holding the second operand.

### Operation

VMLA multiplies each element in a vector by a scalar, and accumulates the results into the corresponding elements of the destination vector.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.62 VMLAL (by scalar)

Vector Multiply by scalar and Accumulate Long.

### Syntax

`VMLAL{cond}.datatype Qd, Dn, Dm[x]`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S16, S32, U16, or U32

*Qd, Dn*

are the destination vector and the first operand vector, for a long operation.

*Dm[x]*

is the scalar holding the second operand.

### Operation

VMLAL multiplies each element in a vector by a scalar, and accumulates the results into the corresponding elements of the destination vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.63 VMLAL

Vector Multiply Accumulate Long.

### Syntax

`VMLAL{cond}.datatype Qd, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

### Operation

VMLAL multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

### Related concepts

[B1.11 Polynomial arithmetic over {0,1} on page B1-101](#)

## C3.64 VMLS (by scalar)

Vector Multiply by scalar and Subtract.

### Syntax

`VMLS{cond}.datatype {Qd}, Qn, Dm[x]`

`VMLS{cond}.datatype {Dd}, Dn, Dm[x]`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or F32.

*Qd, Qn*

are the destination vector and the first operand vector, for a quadword operation.

*Dd, Dn*

are the destination vector and the first operand vector, for a doubleword operation.

*Dm[x]*

is the scalar holding the second operand.

### Operation

VMLS multiplies each element in a vector by a scalar, subtracts the results from the corresponding elements of the destination vector, and places the final results in the destination vector.

#### *Related reference*

[C1.9 Condition code suffixes on page C1-142](#)

## C3.65 VMLS

Vector Multiply Subtract.

### Syntax

VMLS{cond}.datatype {Qd}, Qn, Qm

VMLS{cond}.datatype {Dd}, Dn, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, F32.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VMLS multiplies corresponding elements in two vectors, subtracts the results from corresponding elements of the destination vector, and places the final results in the destination vector.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.66 VMLSL

Vector Multiply Subtract Long.

### Syntax

`VMLSL{cond}.datatype Qd, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

### Operation

VMLSL multiplies corresponding elements in two vectors, subtracts the results from corresponding elements of the destination vector, and places the final results in the destination vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.67 VMLSL (by scalar)

Vector Multiply by scalar and Subtract Long.

### Syntax

`VMLSL{cond}.datatype Qd, Dn, Dm[x]`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S16, S32, U16, or U32.

*Qd, Dn*

are the destination vector and the first operand vector, for a long operation.

*Dm[x]*

is the scalar holding the second operand.

### Operation

VMLSL multiplies each element in a vector by a scalar, subtracts the results from the corresponding elements of the destination vector, and places the final results in the destination vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.68 VMOV (immediate)

Vector Move.

### Syntax

`VMOV{cond}.datatype Qd, #imm`

`VMOV{cond}.datatype Dd, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, I64, or F32.

*Qd* or *Dd*

is the Advanced SIMD register for the result.

*imm*

is an immediate value of the type specified by *datatype*. This is replicated to fill the destination register.

### Operation

VMOV replicates an immediate value in every element of the destination register.

Table C3-7 Available immediate values in VMOV (immediate)

datatype	imm
I8	0xXY
I16	0x00XY, 0xXY00
I32	0x000000XY, 0x0000XY00, 0x00XY0000, 0XY000000
	0x0000XYFF, 0x00XYFFFF
I64	byte masks, 0xGGHHJJKKLLMMNNPP <sup>am</sup>
F32	floating-point numbers <sup>an</sup>

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

<sup>am</sup> Each of 0xGG, 0xHH, 0xJJ, 0xKK, 0xLL, 0xMM, 0xNN, and 0xPP must be either 0x00 or 0xFF.  
<sup>an</sup> Any number that can be expressed as  $+/-n * 2^{-r}$ , where  $n$  and  $r$  are integers,  $16 \leq n \leq 31$ ,  $0 \leq r \leq 7$ .

## C3.69 VMOV (register)

Vector Move.

### Syntax

`VMOV{cond}{.datatype} Qd, Qm`

`VMOV{cond}{.datatype} Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional datatype. The assembler ignores *datatype*.

*Qd, Qm*

specifies the destination vector and the source vector, for a quadword operation.

*Dd, Dm*

specifies the destination vector and the source vector, for a doubleword operation.

### Operation

VMOV copies the contents of the source register into the destination register.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.70 VMOV (between two general-purpose registers and a 64-bit extension register)

Transfer contents between two general-purpose registers and a 64-bit extension register.

### Syntax

VMOV{cond} *Dm*, *Rd*, *Rn*

VMOV{cond} *Rd*, *Rn*, *Dm*

where:

*cond*

is an optional condition code.

*Dm*

is a 64-bit extension register.

*Rd*, *Rn*

are the general-purpose registers. *Rd* and *Rn* must not be PC.

### Operation

VMOV *Dm*, *Rd*, *Rn* transfers the contents of *Rd* into the low half of *Dm*, and the contents of *Rn* into the high half of *Dm*.

VMOV *Rd*, *Rn*, *Dm* transfers the contents of the low half of *Dm* into *Rd*, and the contents of the high half of *Dm* into *Rn*.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.71 VMOV (between a general-purpose register and an Advanced SIMD scalar)

Transfer contents between a general-purpose register and an Advanced SIMD scalar.

### Syntax

`VMOV{cond}{.size} Dn[x], Rd`

`VMOV{cond}{.datatype} Rd, Dn[x]`

where:

*cond*

is an optional condition code.

*size*

the data size. Can be 8, 16, or 32. If omitted, *size* is 32.

*datatype*

the data type. Can be U8, S8, U16, S16, or 32. If omitted, *datatype* is 32.

*Dn[x]*

is the Advanced SIMD scalar.

*Rd*

is the general-purpose register. *Rd* must not be PC.

### Operation

`VMOV Dn[x], Rd` transfers the contents of the least significant byte, halfword, or word of *Rd* into *Dn[x]*.

`VMOV Rd, Dn[x]` transfers the contents of *Dn[x]* into the least significant byte, halfword, or word of *Rd*. The remaining bits of *Rd* are either zero or sign extended.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.72 VMOVL

Vector Move Long.

### Syntax

`VMOVL{cond}.datatype Qd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Dm*

specifies the destination vector and the operand vector.

### Operation

VMOVL takes each element in a doubleword vector, sign or zero extends them to twice their original length, and places the results in a quadword vector.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.73 VMOVN

Vector Move and Narrow.

### Syntax

`VMOVN{cond}.datatype Dd, Qm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or I64.

*Dd, Qm*

specifies the destination vector and the operand vector.

### Operation

VMOVN copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.74 VMOV2

Pseudo-instruction that generates an immediate value and places it in every element of an Advanced SIMD vector, without loading a value from a literal pool.

### Syntax

`VMOV2{cond}.datatype Qd, #constant`

`VMOV2{cond}.datatype Dd, #constant`

where:

*datatype*

must be one of:

- I8, I16, I32, or I64.
- S8, S16, S32, or S64.
- U8, U16, U32, or U64.
- F32.

*cond*

is an optional condition code.

*Qd* or *Dd*

is the extension register to be loaded.

*constant*

is an immediate value of the appropriate type for *datatype*.

### Operation

VMOV2 can generate any 16-bit immediate value, and a restricted range of 32-bit and 64-bit immediate values.

VMOV2 is a pseudo-instruction that always assembles to exactly two instructions. It typically assembles to a VMOV or VMVN instruction, followed by a VBIC or VORR instruction.

### Related reference

[C3.68 VMOV \(immediate\) on page C3-517](#)

[C3.16 VBIC \(immediate\) on page C3-462](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.75 VMRS

Transfer contents from an Advanced SIMD system register to a general-purpose register.

### Syntax

VMRS{cond} Rd, extsysreg

where:

*cond*

is an optional condition code.

*extsysreg*

is the Advanced SIMD and floating-point system register, usually FPSCR, FPSID, or FPEXC.

*Rd*

is the general-purpose register. *Rd* must not be PC.

It can be APSR\_nzcv, if *extsysreg* is FPSCR. In this case, the floating-point status flags are transferred into the corresponding flags in the special-purpose APSR.

### Usage

The VMRS instruction transfers the contents of *extsysreg* into *Rd*.

— Note —

The instruction stalls the processor until all current Advanced SIMD or floating-point operations complete.

### Example

```
VMRS    r2,FPCID
VMRS    APSR_nzcv, FPSCR      ; transfer FP status register to the
                                ; special-purpose APSR
```

### Related reference

[B1.17 Advanced SIMD system registers in AArch32 state](#) on page B1-107

[C1.9 Condition code suffixes](#) on page C1-142

[C4.26 VMRS \(floating-point\)](#) on page C4-627

## C3.76 VMSR

Transfer contents of a general-purpose register to an Advanced SIMD system register.

### Syntax

VMSR{cond} extsysreg, Rd

where:

*cond*

is an optional condition code.

*extsysreg*

is the Advanced SIMD and floating-point system register, usually FPSCR, FPSID, or FPEXC.

*Rd*

is the general-purpose register. *Rd* must not be PC.

It can be APSR\_nzcv, if *extsysreg* is FPSCR. In this case, the floating-point status flags are transferred into the corresponding flags in the special-purpose APSR.

### Usage

The VMSR instruction transfers the contents of *Rd* into *extsysreg*.

— Note —

The instruction stalls the processor until all current Advanced SIMD operations complete.

### Example

VMSR      FPSCR, r4

#### Related reference

[B1.17 Advanced SIMD system registers in AArch32 state](#) on page B1-107

[C1.9 Condition code suffixes](#) on page C1-142

[C4.27 VMSR \(floating-point\)](#) on page C4-628

## C3.77 VMUL

Vector Multiply.

### Syntax

`VMUL{cond}.datatype {Qd}, Qn, Qm`

`VMUL{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, F32, or P8.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VMUL multiplies corresponding elements in two vectors, and places the results in the destination vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.78 VMUL (by scalar)

Vector Multiply by scalar.

### Syntax

`VMUL{cond}.datatype {Qd}, Qn, Dm[x]`

`VMUL{cond}.datatype {Dd}, Dn, Dm[x]`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or F32.

*Qd, Qn*

are the destination vector and the first operand vector, for a quadword operation.

*Dd, Dn*

are the destination vector and the first operand vector, for a doubleword operation.

*Dm[x]*

is the scalar holding the second operand.

### Operation

VMUL multiplies each element in a vector by a scalar, and places the results in the destination vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.79 VMULL

Vector Multiply Long

### Syntax

`VMULL{cond}.datatype Qd, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of U8, U16, U32, S8, S16, S32, or P8.

*Qd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

### Operation

VMULL multiplies corresponding elements in two vectors, and places the results in the destination vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.80 VMULL (by scalar)

Vector Multiply Long by scalar

### Syntax

`VMULL{cond}.datatype Qd, Dn, Dm[x]`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S16, S32, U16, or U32.

*Qd, Dn*

are the destination vector and the first operand vector, for a long operation.

*Dm[x]*

is the scalar holding the second operand.

### Operation

VMULL multiplies each element in a vector by a scalar, and places the results in the destination vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.81 VMVN (register)

Vector Move NOT (register).

### Syntax

`VMVN{cond}{.datatype} Qd, Qm`

`VMVN{cond}{.datatype} Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional datatype. The assembler ignores *datatype*.

*Qd, Qm*

specifies the destination vector and the source vector, for a quadword operation.

*Dd, Dm*

specifies the destination vector and the source vector, for a doubleword operation.

### Operation

VMVN inverts the value of each bit from the source register and places the results into the destination register.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.82 VMVN (immediate)

Vector Move NOT (immediate).

### Syntax

`VMVN{cond}.datatype Qd, #imm`

`VMVN{cond}.datatype Dd, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, I64, or F32.

*Qd* or *Dd*

is the Advanced SIMD register for the result.

*i.m*m

is an immediate value of the type specified by *datatype*. This is replicated to fill the destination register.

### Operation

VMVN inverts the value of each bit from an immediate value and places the results into each element in the destination register.

**Table C3-8 Available immediate values in VMVN (immediate)**

datatype	imm
I8	-
I16	0xFFXY, 0XXYFF
I32	0xFFFFFFFFXY, 0xFFFFXYFF, 0xFFXYFFFF, 0XXYFFFFFF
	0xFFFFXY00, 0FFXY0000
I64	-
F32	-

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C3.83 VNEG

Vector Negate.

### Syntax

VNEG{cond}.datatype Qd, Qm

VNEG{cond}.datatype Dd, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, or F32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VNEG negates each element in a vector, and places the results in a second vector. (The floating-point version only inverts the sign bit.)

#### Related reference

[C4.29 VNEG \(floating-point\) on page C4-630](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.84 VORN (register)

Vector bitwise OR NOT (register).

### Syntax

`VORN{cond}{.datatype} {Qd}, Qn, Qm`

`VORN{cond}{.datatype} {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional data type. The assembler ignores *datatype*.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VORN performs a bitwise logical OR complement between two registers, and places the results in the destination register.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.85 VORN (immediate)

Vector bitwise OR NOT (immediate) pseudo-instruction.

### Syntax

VORN{cond}.datatype Qd, #imm

VORN{cond}.datatype Dd, #imm

where:

*cond*

is an optional condition code.

*datatype*

must be either I8, I16, I32, or I64.

*Qd* or *Dd*

is the Advanced SIMD register for the result.

*i*.imm

is the immediate value.

### Operation

VORN takes each element of the destination vector, performs a bitwise OR complement with an immediate value, and returns the results in the destination vector.

---

#### Note

---

On disassembly, this pseudo-instruction is disassembled to a corresponding VORR instruction, with a complementary immediate value.

---

### Immediate values

If *datatype* is I16, the immediate value must have one of the following forms:

- 0xFFXY.
- 0xXYFF.

If *datatype* is I32, the immediate value must have one of the following forms:

- 0xFFFFFFFXY.
- 0xFFFFXXYFF.
- 0xFFXYFFFF.
- 0xXYFFFFFF.

### Related reference

[C3.87 VORR \(immediate\) on page C3-536](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.86 VORR (register)

Vector bitwise OR (register).

### Syntax

VORR{cond}{.datatype} {Qd}, Qn, Qm

VORR{cond}{.datatype} {Dd}, Dn, Dm

where:

*cond*

is an optional condition code.

*datatype*

is an optional data type. The assembler ignores *datatype*.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

---

### Note

---

VORR with the same register for both operands is a VMOV instruction. You can use VORR in this way, but disassembly of the resulting code produces the VMOV syntax.

---

### Operation

VORR performs a bitwise logical OR between two registers, and places the result in the destination register.

### Related reference

[C3.69 VMOV \(register\) on page C3-518](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.87 VORR (immediate)

Vector bitwise OR immediate.

### Syntax

`VORR{cond}.datatype Qd, #imm`

`VORR{cond}.datatype Dd, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be either I8, I16, I32, or I64.

*Qd* or *Dd*

is the Advanced SIMD register for the source and result.

*imm*

is the immediate value.

### Operation

VORR takes each element of the destination vector, performs a bitwise logical OR with an immediate value, and places the results in the destination vector.

### Immediate values

You can either specify *imm* as a pattern which the assembler repeats to fill the destination register, or you can directly specify the immediate value (that conforms to the pattern) in full. The pattern for *imm* depends on the datatype, as shown in the following table:

**Table C3-9 Patterns for immediate value in VORR (immediate)**

I16	I32
0x00XY	0x000000XY
0xXY00	0x0000XY00
-	0x00XY0000
-	0XXY000000

If you use the I8 or I64 datatypes, the assembler converts it to either the I16 or I32 instruction to match the pattern of *imm*. If the immediate value does not match any of the patterns in the preceding table, the assembler generates an error.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.88 VPADAL

Vector Pairwise Add and Accumulate Long.

### Syntax

VPADAL{cond}.datatype Qd, Qm

VPADAL{cond}.datatype Dd, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword instruction.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword instruction.

### Operation

VPADAL adds adjacent pairs of elements of a vector, and accumulates the absolute values of the results into the elements of the destination vector.

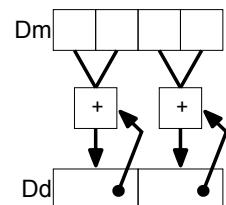


Figure C3-3 Example of operation of VPADAL (in this case for data type S16)

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.89 VPADD

Vector Pairwise Add.

### Syntax

VPADD{cond}.datatype {Dd}, Dn, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, or F32.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector.

### Operation

VPADD adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

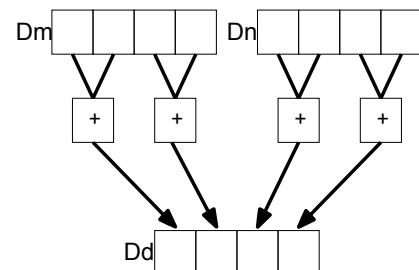


Figure C3-4 Example of operation of VPADD (in this case, for data type I16)

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.90 VPADDL

Vector Pairwise Add Long.

### Syntax

VPADDL{cond}.datatype Qd, Qm

VPADDL{cond}.datatype Dd, Dm

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword instruction.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword instruction.

### Operation

VPADDL adds adjacent pairs of elements of a vector, sign or zero extends the results to twice their original width, and places the final results in the destination vector.

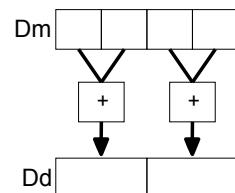


Figure C3-5 Example of operation of doubleword VPADDL (in this case, for data type S16)

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.91 VPMAX and VPMIN

Vector Pairwise Maximum, Vector Pairwise Minimum.

### Syntax

`VPop{cond}.datatype Dd, Dn, Dm`

where:

*op*

must be either MAX or MIN.

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, U32, or F32.

*Dd, Dn, Dm*

are the destination doubleword vector, the first operand doubleword vector, and the second operand doubleword vector.

### Operation

VPMAX compares adjacent pairs of elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector. Operands and results must be doubleword vectors.

VPMIN compares adjacent pairs of elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector. Operands and results must be doubleword vectors.

### Floating-point maximum and minimum

$\max(+0.0, -0.0) = +0.0$ .

$\min(+0.0, -0.0) = -0.0$

If any input is a NaN, the corresponding result element is the default NaN.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.92 VPOP

Pop extension registers from the stack.

### Syntax

`VPOP{cond} Registers`

where:

*cond*

is an optional condition code.

*Registers*

is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify D or Q registers, but they must not be mixed. The number of registers must not exceed 16 D registers, or 8 Q registers. If Q registers are specified, on disassembly they are shown as D registers.

---

### Note

---

`VPOP Registers` is equivalent to `VLDM sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPOP`.

---

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

[C3.93 VPUSH](#) on page C3-542

[C4.33 VPOP \(floating-point\)](#) on page C4-634

## C3.93 VPUSH

Push extension registers onto the stack.

### Syntax

`VPUSH{cond} Registers`

where:

*cond*

is an optional condition code.

*Registers*

is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify D or Q registers, but they must not be mixed. The number of registers must not exceed 16 D registers, or 8 Q registers. If Q registers are specified, on disassembly they are shown as D registers.

---

### Note

---

`VPUSH Registers` is equivalent to `VSTMDB sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPUSH`.

---

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

[C3.92 VPOP on page C3-541](#)

[C4.34 VPUSH \(floating-point\) on page C4-635](#)

## C3.94 VQABS

Vector Saturating Absolute.

### Syntax

`VQABS{cond}.datatype Qd, Qm`

`VQABS{cond}.datatype Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, or S32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VQABS takes the absolute value of each element in a vector, and places the results in a second vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.95 VQADD

Vector Saturating Add.

### Syntax

`VQADD{cond}.datatype {Qd}, Qn, Qm`

`VQADD{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VQADD adds corresponding elements in two vectors, and places the results in the destination vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.96 VQDMLAL and VQDMLSL (by vector or by scalar)

Vector Saturating Doubling Multiply Accumulate Long, Vector Saturating Doubling Multiply Subtract Long.

### Syntax

`VQDopL{cond}.datatype Qd, Dn, Dm`

`VQDopL{cond}.datatype Qd, Dn, Dm[x]`

where:

*op*

must be one of:

**MLA**

Multiply Accumulate.

**MLS**

Multiply Subtract.

*cond*

is an optional condition code.

*datatype*

must be either S16 or S32.

*Qd, Dn*

are the destination vector and the first operand vector.

*Dm*

is the vector holding the second operand, for a *by vector* operation.

*Dm[x]*

is the scalar holding the second operand, for a *by scalar* operation.

### Operation

These instructions multiply their operands and double the results. VQDMLAL adds the results to the values in the destination register. VQDMLSL subtracts the results from the values in the destination register.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.97 VQDMULH (by vector or by scalar)

Vector Saturating Doubling Multiply Returning High Half.

### Syntax

```
VQDMULH{cond}.datatype {Qd}, Qn, Qm  
VQDMULH{cond}.datatype {Dd}, Dn, Dm  
VQDMULH{cond}.datatype {Qd}, Qn, Dm[x]  
VQDMULH{cond}.datatype {Dd}, Dn, Dm[x]
```

where:

*cond*

is an optional condition code.

*datatype*

must be either S16 or S32.

*Qd, Qn*

are the destination vector and the first operand vector, for a quadword operation.

*Dd, Dn*

are the destination vector and the first operand vector, for a doubleword operation.

*Qm or Dm*

is the vector holding the second operand, for a *by vector* operation.

*Dm[x]*

is the scalar holding the second operand, for a *by scalar* operation.

### Operation

VQDMULH multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector.

The second operand can be a scalar instead of a vector.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs. Each result is truncated.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.98 VQDMULL (by vector or by scalar)

Vector Saturating Doubling Multiply Long.

### Syntax

`VQDMULL{cond}.datatype Qd, Dn, Dm`

`VQDMULL{cond}.datatype Qd, Dn, Dm[x]`

where:

*cond*

is an optional condition code.

*datatype*

must be either S16 or S32.

*Qd, Dn*

are the destination vector and the first operand vector.

*Dm*

is the vector holding the second operand, for a *by vector* operation.

*Dm[x]*

is the scalar holding the second operand, for a *by scalar* operation.

### Operation

VQDMULL multiplies corresponding elements in two vectors, doubles the results and places the results in the destination register.

The second operand can be a scalar instead of a vector.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.99 VQMOVN and VQMOVUN

Vector Saturating Move and Narrow.

### Syntax

`VQMOVN{cond}.datatype Dd, Qm`

`VQMOVUN{cond}.datatype Dd, Qm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of:

**S16, S32, S64**

for VQMOVN or VQMOVUN.

**U16, U32, U64**

for VQMOVN.

*Dd, Qm*

specifies the destination vector and the operand vector.

### Operation

VQMOVN copies each element of the operand vector to the corresponding element of the destination vector. The result element is half the width of the operand element, and values are saturated to the result width. The results are the same type as the operands.

VQMOVUN copies each element of the operand vector to the corresponding element of the destination vector. The result element is half the width of the operand element, and values are saturated to the result width. The elements in the operand are signed and the elements in the result are unsigned.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.100 VQNEG

Vector Saturating Negate.

### Syntax

`VQNEG{cond}.datatype Qd, Qm`

`VQNEG{cond}.datatype Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, or S32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VQNEG negates each element in a vector, and places the results in a second vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.101 VQRDMULH (by vector or by scalar)

Vector Saturating Rounding Doubling Multiply Returning High Half.

### Syntax

```
VQRDMULH{cond}.datatype {Qd}, Qn, Qm  
VQRDMULH{cond}.datatype {Dd}, Dn, Dm  
VQRDMULH{cond}.datatype {Qd}, Qn, Dm[x]  
VQRDMULH{cond}.datatype {Dd}, Dn, Dm[x]
```

where:

*cond*

is an optional condition code.

*datatype*

must be either S16 or S32.

*Qd, Qn*

are the destination vector and the first operand vector, for a quadword operation.

*Dd, Dn*

are the destination vector and the first operand vector, for a doubleword operation.

*Qm or Dm*

is the vector holding the second operand, for a *by vector* operation.

*Dm[x]*

is the scalar holding the second operand, for a *by scalar* operation.

### Operation

VQRDMULH multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector.

The second operand can be a scalar instead of a vector.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs. Each result is rounded.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.102 VQRSHL (by signed variable)

Vector Saturating Rounding Shift Left by signed variable.

### Syntax

`VQRSHL{cond}.datatype {Qd}, Qm, Qn`

`VQRSHL{cond}.datatype {Dd}, Dm, Dn`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qm, Qn*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dm, Dn*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VQRSHL takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a rounding right shift.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.103 VQRSHRN and VQRSHRUN (by immediate)

Vector Saturating Shift Right, Narrow, by immediate value, with Rounding.

### Syntax

`VQRSHR{U}N{cond}.datatype Dd, Qm, #imm`

where:

**U**

if present, indicates that the results are unsigned, although the operands are signed. Otherwise, the results are the same type as the operands.

*cond*

is an optional condition code.

*datatype*

must be one of:

**I16, I32, I64**

for VQRSHRN or VQRSHRUN. Only a #0 immediate is permitted with these datatypes.

**S16, S32, S64**

for VQRSHRN or VQRSHRUN.

**U16, U32, U64**

for VQRSHRN only.

*Dd, Qm*

are the destination vector and the operand vector.

*imm*

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

**Table C3-10 Available immediate ranges in VQRSHRN and VQRSHRUN (by immediate)**

datatype	imm range
S16 or U16	0 to 8
S32 or U32	0 to 16
S64 or U64	0 to 32

### Operation

`VQRSHR{U}N` takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the results in a doubleword vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Results are rounded.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.104 VQSHL (by signed variable)

Vector Saturating Shift Left by signed variable.

### Syntax

`VQSHL{cond}.datatype {Qd}, Qm, Qn`

`VQSHL{cond}.datatype {Dd}, Dm, Dn`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qm, Qn*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dm, Dn*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VQSHL takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a truncating right shift.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.105 VQSHL and VQSHLU (by immediate)

Vector Saturating Shift Left.

### Syntax

`VQSHL{U}{cond}.datatype {Qd}, Qm, #imm`

`VQSHL{U}{cond}.datatype {Dd}, Dm, #imm`

where:

**U**

only permitted if Q is also present. Indicates that the results are unsigned even though the operands are signed.

*cond*

is an optional condition code.

*datatype*

must be one of :

**S8, S16, S32, S64**

for VQSHL or VQSHLU.

**U8, U16, U32, U64**

for VQSHL only.

*Qd, Qm*

are the destination and operand vectors, for a quadword operation.

*Dd, Dm*

are the destination and operand vectors, for a doubleword operation.

*imm*

is the immediate value specifying the size of the shift, in the range 0 to (size(*datatype*) – 1).  
The ranges are shown in the following table:

**Table C3-11 Available immediate ranges in VQSHL and VQSHLU (by immediate)**

datatype	imm range
S8 or U8	0 to 7
S16 or U16	0 to 15
S32 or U32	0 to 31
S64 or U64	0 to 63

### Operation

VQSHL and VQSHLU instructions take each element in a vector of integers, left shift them by an immediate value, and place the results in the destination vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.106 VQSHRN and VQSHRUN (by immediate)

Vector Saturating Shift Right, Narrow, by immediate value.

### Syntax

`VQSHR{U}N{cond}.datatype Dd, Qm, #imm`

where:

**U**

if present, indicates that the results are unsigned, although the operands are signed. Otherwise, the results are the same type as the operands.

*cond*

is an optional condition code.

*datatype*

must be one of:

**I16, I32, I64**

for VQSHRN or VQSHRUN. Only a #0 immediate is permitted with these datatypes.

**S16, S32, S64**

for VQSHRN or VQSHRUN.

**U16, U32, U64**

for VQSHRN only.

*Dd, Qm*

are the destination vector and the operand vector.

*imm*

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

**Table C3-12 Available immediate ranges in VQSHRN and VQSHRUN (by immediate)**

datatype	imm range
S16 or U16	0 to 8
S32 or U32	0 to 16
S64 or U64	0 to 32

### Operation

`VQSHR{U}N` takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the results in a doubleword vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Results are truncated.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.107 VQSUB

Vector Saturating Subtract.

### Syntax

`VQSUB{cond}.datatype {Qd}, Qn, Qm`

`VQSUB{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VQSUB subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.108 VRADDHN

Vector Rounding Add and Narrow, selecting High half.

### Syntax

`VRADDHN{cond}.datatype Dd, Qn, Qm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or I64.

*Dd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector.

### Operation

VRADDHN adds corresponding elements in two quadword vectors, selects the most significant halves of the results, and places the final results in the destination doubleword vector. Results are rounded.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.109 VRECPE

Vector Reciprocal Estimate.

### Syntax

`VRECPE{cond}.datatype Qd, Qm`

`VRECPE{cond}.datatype Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be either U32 or F32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VRECPE finds an approximate reciprocal of each element in a vector, and places the results in a second vector.

### Results for out-of-range inputs

The following table shows the results where input values are out of range:

**Table C3-13 Results for out-of-range inputs in VRECPE**

	<b>Operand element</b>	<b>Result element</b>
Integer	$\leq 0x7FFFFFFF$	$0xFFFFFFFF$
Floating-point	NaN	Default NaN
	Negative 0, Negative Denormal	Negative Infinity <sup>ao</sup>
	Positive 0, Positive Denormal	Positive Infinity <sup>ao</sup>
	Positive infinity	Positive 0
	Negative infinity	Negative 0

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

<sup>ao</sup> The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

## C3.110 VRECPS

Vector Reciprocal Step.

### Syntax

`VRECPS{cond}.F32 {Qd}, Qn, Qm`

`VRECPS{cond}.F32 {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VRECPS multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the results from 2, and places the final results into the elements of the destination vector.

The Newton-Raphson iteration:

$$x_{n+1} = x_n (2 - dx_n)$$

converges to  $(1/d)$  if  $x_0$  is the result of VRECPE applied to  $d$ .

### Results for out-of-range inputs

The following table shows the results where input values are out of range:

Table C3-14 Results for out-of-range inputs in VRECPS

1st operand element	2nd operand element	Result element
NaN	-	Default NaN
-	NaN	Default NaN
$\pm 0.0$ or denormal	$\pm \infty$	2.0
$\pm \infty$	$\pm 0.0$ or denormal	2.0

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.111 VREV16, VREV32, and VREV64

Vector Reverse within halfwords, words, or doublewords.

### Syntax

`VREVn{cond}.size Qd, Qm`

`VREVn{cond}.size Dd, Dm`

where:

*n*

must be one of 16, 32, or 64.

*cond*

is an optional condition code.

*size*

must be one of 8, 16, or 32, and must be less than *n*.

*Qd, Qm*

specifies the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

specifies the destination vector and the operand vector, for a doubleword operation.

### Operation

VREV16 reverses the order of 8-bit elements within each halfword of the vector, and places the result in the corresponding destination vector.

VREV32 reverses the order of 8-bit or 16-bit elements within each word of the vector, and places the result in the corresponding destination vector.

VREV64 reverses the order of 8-bit, 16-bit, or 32-bit elements within each doubleword of the vector, and places the result in the corresponding destination vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.112 VRHADD

Vector Rounding Halving Add.

### Syntax

`VRHADD{cond}.datatype {Qd}, Qn, Qm`

`VRHADD{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VRHADD adds corresponding elements in two vectors, shifts each result right one bit, and places the results in the destination vector. Results are rounded.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.113 VRSHL (by signed variable)

Vector Rounding Shift Left by signed variable.

### Syntax

`VRSHL{cond}.datatype {Qd}, Qm, Qn`

`VRSHL{cond}.datatype {Dd}, Dm, Dn`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qm, Qn*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dm, Dn*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VRSHL takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a rounding right shift.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.114 VRSHR (by immediate)

Vector Rounding Shift Right by immediate value.

### Syntax

`VRSHR{cond}.datatype {Qd}, Qm, #imm`

`VRSHR{cond}.datatype {Dd}, Dm, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

*iimm*

is the immediate value specifying the size of the shift, in the range 0 to (size(*datatype*)). The ranges are shown in the following table:

**Table C3-15 Available immediate ranges in VRSHR (by immediate)**

datatype	imm range
S8 or U8	0 to 8
S16 or U16	0 to 16
S32 or U32	0 to 32
S64 or U64	0 to 64

VRSHR with an immediate value of zero is a pseudo-instruction for VORR.

### Operation

VRSHR takes each element in a vector, right shifts them by an immediate value, and places the results in the destination vector. The results are rounded.

### Related reference

[C3.86 VORR \(register\) on page C3-535](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.115 VRSHRN (by immediate)

Vector Rounding Shift Right, Narrow, by immediate value.

### Syntax

`VRSHRN{cond}.datatype Dd, Qm, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or I64.

*Dd, Qm*

are the destination vector and the operand vector.

*imm*

is the immediate value specifying the size of the shift, in the range 0 to  $(\text{size}(\text{datatype})/2)$ . The ranges are shown in the following table:

**Table C3-16 Available immediate ranges in VRSHRN (by immediate)**

datatype	imm range
I16	0 to 8
I32	0 to 16
I64	0 to 32

VRSHRN with an immediate value of zero is a pseudo-instruction for VMOVN.

### Operation

VRSHRN takes each element in a quadword vector, right shifts them by an immediate value, and places the results in a doubleword vector. The results are rounded.

### Related reference

[C3.73 VMOVN on page C3-522](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.116 VRINT

VRINT (Vector Round to Integer) rounds each floating-point element in a vector to integer, and places the results in the destination vector.

The resulting integers are represented in floating-point format.

— Note —

This instruction is supported only in Armv8.

### Syntax

`VRINTmode.F32.F32 Qd, Qm`

`VRINTmode.F32.F32 Dd, Dm`

where:

*mode*

must be one of:

**A**

meaning round to nearest, ties away from zero. This cannot generate an Inexact exception, even if the result is not exact.

**N**

meaning round to nearest, ties to even. This cannot generate an Inexact exception, even if the result is not exact.

**X**

meaning round to nearest, ties to even, generating an Inexact exception if the result is not exact.

**P**

meaning round towards plus infinity. This cannot generate an Inexact exception, even if the result is not exact.

**M**

meaning round towards minus infinity. This cannot generate an Inexact exception, even if the result is not exact.

**Z**

meaning round towards zero. This cannot generate an Inexact exception, even if the result is not exact.

*Qd, Qm*

specifies the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

specifies the destination and operand vectors, for a doubleword operation.

### Notes

You cannot use VRINT inside an IT block.

## C3.117 VRSQRTE

Vector Reciprocal Square Root Estimate.

### Syntax

`VRSQRTE{cond}.datatype Qd, Qm`

`VRSQRTE{cond}.datatype Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be either U32 or F32.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

### Operation

VRSQRTE finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

### Results for out-of-range inputs

The following table shows the results where input values are out of range:

Table C3-17 Results for out-of-range inputs in VRSQRTE

	Operand element	Result element
Integer	$\leq 0x3FFFFFFF$	0xFFFFFFFF
Floating-point	NaN, Negative Normal, Negative Infinity	Default NaN
	Negative 0, Negative Denormal	Negative Infinity <sup>ap</sup>
	Positive 0, Positive Denormal	Positive Infinity <sup>ap</sup>
	Positive infinity	Positive 0
		Negative 0

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

<sup>ap</sup> The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

## C3.118 VRSQRTS

Vector Reciprocal Square Root Step.

### Syntax

`VRSQRTS{cond}.F32 {Qd}, Qn, Qm`

`VRSQRTS{cond}.F32 {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VRSQRTS multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the results from three, divides these results by two, and places the final results into the elements of the destination vector.

The Newton-Raphson iteration:

$$x_{n+1} = x_n (3 - dx_n^2)/2$$

converges to  $(1/\sqrt{d})$  if  $x_0$  is the result of VRSQRTE applied to  $d$ .

### Results for out-of-range inputs

The following table shows the results where input values are out of range:

Table C3-18 Results for out-of-range inputs in VRSQRTS

1st operand element	2nd operand element	Result element
NaN	-	Default NaN
-	NaN	Default NaN
$\pm 0.0$ or denormal	$\pm \infty$	1.5
$\pm \infty$	$\pm 0.0$ or denormal	1.5

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.119 VRSRA (by immediate)

Vector Rounding Shift Right by immediate value and Accumulate.

### Syntax

`VRSRA{cond}.datatype {Qd}, Qm, #imm`

`VRSRA{cond}.datatype {Dd}, Dm, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

*iimm*

is the immediate value specifying the size of the shift, in the range 1 to (size(*datatype*)). The ranges are shown in the following table:

**Table C3-19 Available immediate ranges in VRSRA (by immediate)**

datatype	imm range
S8 or U8	1 to 8
S16 or U16	1 to 16
S32 or U32	1 to 32
S64 or U64	1 to 64

### Operation

VRSRA takes each element in a vector, right shifts them by an immediate value, and accumulates the results into the destination vector. The results are rounded.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C3.120 VRSUBHN

Vector Rounding Subtract and Narrow, selecting High half.

### Syntax

`VRSUBHN{cond}.datatype Dd, Qn, Qm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or I64.

*Dd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector.

### Operation

VRSUBHN subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, selects the most significant halves of the results, and places the final results in the destination doubleword vector. Results are rounded.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.121 VSDOT (vector)

Dot Product vector form with signed integers.

### Syntax

`VSDOT{q}.S8 Dd, Dn, Dm ; 64-bit SIMD vector`

`VSDOT{q}.S8 Qd, Qn, Qm ; A1 128-bit SIMD vector FP/SIMD registers (A32)`

Where:

- q** Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page [C2-161](#).
- Dd** Is the 64-bit name of the SIMD and FP destination register.
- Dn** Is the 64-bit name of the first SIMD and FP source register.
- Dm** Is the 64-bit name of the second SIMD and FP source register.
- Qd** Is the 128-bit name of the SIMD and FP destination register.
- Qn** Is the 128-bit name of the first SIMD and FP source register.
- Qm** Is the 128-bit name of the second SIMD and FP source register.

### Architectures supported

Supported in Armv8.2 and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

### Usage

Dot Product vector form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

#### Note

ID\_ISAR6.DP indicates whether this instruction is supported in the T32 and A32 instruction sets.

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page [C3-445](#)

## C3.122 VSDOT (by element)

Dot Product index form with signed integers.

### Syntax

`VSDOT{q}.S8 Dd, Dn, Dm[index] ; 64-bit SIMD vector`

`VSDOT{q}.S8 Qd, Qn, Dm[index] ; A1 128-bit SIMD vector FP/SIMD registers (A32)`

Where:

- q*** Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page [C2-161](#).
- Dd*** Is the 64-bit name of the SIMD and FP destination register.
- Dn*** Is the 64-bit name of the first SIMD and FP source register.
- Dm*** Is the 64-bit name of the second SIMD and FP source register.
- index*** Is the element index in the range 0 to 1.
- Qd*** Is the 128-bit name of the SIMD and FP destination register.
- Qn*** Is the 128-bit name of the first SIMD and FP source register.

### Architectures supported

Supported in Armv8.2 and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

### Usage

Dot Product index form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

---

#### Note

---

ID\_ISAR6.DP indicates whether this instruction is supported in the T32 and A32 instruction sets.

---

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page [C3-445](#)

## C3.123 VSHL (by immediate)

Vector Shift Left by immediate.

### Syntax

`VSHL{cond}.datatype {Qd}, Qm, #imm`

`VSHL{cond}.datatype {Dd}, Dm, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, or I64.

*Qd, Qm*

are the destination and operand vectors, for a quadword operation.

*Dd, Dm*

are the destination and operand vectors, for a doubleword operation.

*iimm*

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

**Table C3-20 Available immediate ranges in VSHL (by immediate)**

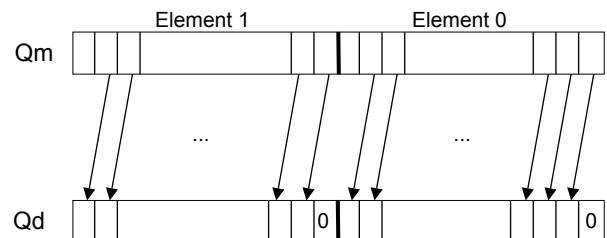
datatype	imm range
I8	0 to 7
I16	0 to 15
I32	0 to 31
I64	0 to 63

### Operation

VSHL takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

Bits shifted out of the left of each element are lost.

The following figure shows the operation of VSHL with two elements and a shift value of one. The least significant bit in each element in the destination vector is set to zero.



**Figure C3-6 Operation of quadword VSHL.I64 Qd, Qm, #1**

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.124 VSHL (by signed variable)

Vector Shift Left by signed variable.

### Syntax

`VSHL{cond}.datatype {Qd}, Qm, Qn`

`VSHL{cond}.datatype {Dd}, Dm, Dn`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qm, Qn*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

*Dd, Dm, Dn*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

### Operation

VSHL takes each element in a vector, shifts them by the value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a truncating right shift.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.125 VSHLL (by immediate)

Vector Shift Left Long.

### Syntax

`VSHLL{cond}.datatype Qd, Dm, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Dm*

are the destination and operand vectors, for a long operation.

*imm*

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

**Table C3-21 Available immediate ranges in VSHLL (by immediate)**

datatype	imm range
S8 or U8	1 to 8
S16 or U16	1 to 16
S32 or U32	1 to 32

0 is permitted, but the resulting code disassembles to VMOVL.

### Operation

VSHLL takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector. Values are sign or zero extended.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.126 VSHR (by immediate)

Vector Shift Right by immediate value.

### Syntax

`VSHR{cond}.datatype {Qd}, Qm, #imm`

`VSHR{cond}.datatype {Dd}, Dm, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

*iimm*

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

**Table C3-22 Available immediate ranges in VSHR (by immediate)**

datatype	imm range
S8 or U8	0 to 8
S16 or U16	0 to 16
S32 or U32	0 to 32
S64 or U64	0 to 64

VSHR with an immediate value of zero is a pseudo-instruction for VORR.

### Operation

VSHR takes each element in a vector, right shifts them by an immediate value, and places the results in the destination vector. The results are truncated.

### Related reference

[C3.86 VORR \(register\) on page C3-535](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.127 VSHRN (by immediate)

Vector Shift Right, Narrow, by immediate value.

### Syntax

`VSHRN{cond}.datatype Dd, Qm, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or I64.

*Dd, Qm*

are the destination vector and the operand vector.

*imm*

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

**Table C3-23 Available immediate ranges in VSHRN (by immediate)**

datatype	imm range
I16	0 to 8
I32	0 to 16
I64	0 to 32

VSHRN with an immediate value of zero is a pseudo-instruction for VMOVN.

### Operation

VSHRN takes each element in a quadword vector, right shifts them by an immediate value, and places the results in a doubleword vector. The results are truncated.

### Related reference

[C3.73 VMOVN on page C3-522](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C3.128 VSLI

Vector Shift Left and Insert.

### Syntax

`VSLI{cond}.size {Qd}, Qm, #imm`

`VSLI{cond}.size {Dd}, Dm, #imm`

where:

*cond*

is an optional condition code.

*size*

must be one of 8, 16, 32, or 64.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

*imm*

is the immediate value specifying the size of the shift, in the range 0 to (*size* – 1).

### Operation

**VSLI** takes each element in a vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost. The following figure shows the operation of **VSLI** with two elements and a shift value of one. The least significant bit in each element in the destination vector is unchanged.

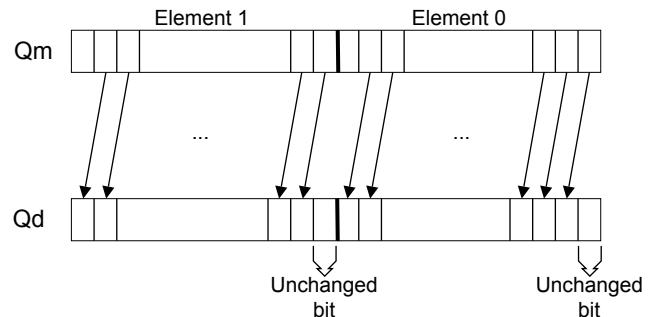


Figure C3-7 Operation of quadword VSLI.64 Qd, Qm, #1

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.129 VSRA (by immediate)

Vector Shift Right by immediate value and Accumulate.

### Syntax

`VSRA{cond}.datatype {Qd}, Qm, #imm`

`VSRA{cond}.datatype {Dd}, Dm, #imm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

*iimm*

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

**Table C3-24 Available immediate ranges in VSRA (by immediate)**

datatype	imm range
S8 or U8	1 to 8
S16 or U16	1 to 16
S32 or U32	1 to 32
S64 or U64	1 to 64

### Operation

VSRA takes each element in a vector, right shifts them by an immediate value, and accumulates the results into the destination vector. The results are truncated.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C3.130 VSRI

Vector Shift Right and Insert.

### Syntax

`VSRI{cond}.size {Qd}, Qm, #imm`

`VSRI{cond}.size {Dd}, Dm, #imm`

where:

*cond*

is an optional condition code.

*size*

must be one of 8, 16, 32, or 64.

*Qd, Qm*

are the destination vector and the operand vector, for a quadword operation.

*Dd, Dm*

are the destination vector and the operand vector, for a doubleword operation.

*imm*

is the immediate value specifying the size of the shift, in the range 1 to *size*.

### Operation

VSRI takes each element in a vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost. The following figure shows the operation of VSRI with a single element and a shift value of two. The two most significant bits in the destination vector are unchanged.

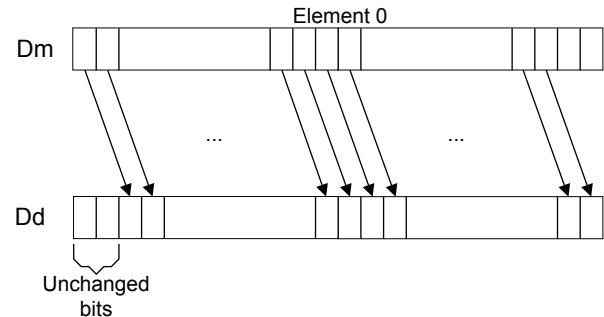


Figure C3-8 Operation of doubleword VSRI.64 Dd, Dm, #2

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C3.131 VSTM

Extension register store multiple.

### Syntax

`VSTMmode{cond} Rn{!}, Registers`

where:

*mode*

must be one of:

**IA**

meaning Increment address After each transfer. IA is the default, and can be omitted.

**DB**

meaning Decrement address Before each transfer.

**EA**

meaning Empty Ascending stack operation. This is the same as IA for stores.

**FD**

meaning Full Descending stack operation. This is the same as DB for stores.

*cond*

is an optional condition code.

*Rn*

is the general-purpose register holding the base address for the transfer.

!

is optional. ! specifies that the updated base address must be written back to *Rn*. If ! is not specified, *mode* must be IA.

*Registers*

is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify D or Q registers, but they must not be mixed. The number of registers must not exceed 16 D registers, or 8 Q registers. If Q registers are specified, on disassembly they are shown as D registers.

---

#### Note

---

`VPUSH Registers` is equivalent to `VSTMDB sp!, Registers`.

You can use either form of this instruction. They both disassemble to VPUSH.

---

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

[C4.38 VSTM \(floating-point\)](#) on page C4-639

## C3.132 VSTn (multiple n-element structures)

Vector Store multiple  $n$ -element structures.

### Syntax

`VSTn{cond}.datatype List, [Rn{@align}]{}!`  
`VSTn{cond}.datatype List, [Rn{@align}], Rm`

where:

*n*

must be one of 1, 2, 3, or 4.

*cond*

is an optional condition code.

*datatype*

see the following table for options.

*List*

is the list of Advanced SIMD registers enclosed in braces, { and }. See the following table for options.

*Rn*

is the general-purpose register containing the base address. *Rn* cannot be PC.

*align*

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the stores have taken place.

*Rm*

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

### Operation

`VSTn` stores multiple  $n$ -element structures to memory from one or more Advanced SIMD registers, with interleaving (unless  $n == 1$ ). Every element of each register is stored.

**Table C3-25 Permitted combinations of parameters for VSTn (multiple n-element structures)**

<i>n</i>	<i>datatype</i>	<i>list aq</i>	<i>align ar</i>	<i>alignment</i>
1	8, 16, 32, or 64	{Dd}	@64	8-byte
		{Dd, D(d+1)}	@64 or @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

*aq* Every register in the list must be in the range D0-D31.

*ar* *align* can be omitted. In this case, standard alignment rules apply.

**Table C3-25 Permitted combinations of parameters for VSTn (multiple n-element structures) (continued)**

<i>n</i>	<i>datatype</i>	<i>list aq</i>	<i>align ar</i>	<i>alignment</i>
2	8, 16, or 32	{Dd, D(d+1)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+2)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
3	8, 16, or 32	{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+2), D(d+4)}	@64	8-byte
4	8, 16, or 32	{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
		{Dd, D(d+2), D(d+4), D(d+6)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

**Related concepts**

[C3.3 Interleaving provided by load and store element and structure instructions](#) on page C3-449

[C3.4 Alignment restrictions in load and store element and structure instructions](#) on page C3-450

**Related reference**

[C1.9 Condition code suffixes](#) on page C1-142

## C3.133 VSTn (single n-element structure to one lane)

Vector Store single  $n$ -element structure to one lane.

### Syntax

`VSTn{cond}.datatype List, [Rn{@align}]{}!`  
`VSTn{cond}.datatype List, [Rn{@align}], Rm`

where:

*n*

must be one of 1, 2, 3, or 4.

*cond*

is an optional condition code.

*datatype*

see the following table.

*List*

is the list of Advanced SIMD registers enclosed in braces, { and }. See the following table for options.

*Rn*

is the general-purpose register containing the base address. *Rn* cannot be PC.

*align*

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the stores have taken place.

*Rm*

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

### Operation

*VSTn* stores one  $n$ -element structure into memory from one or more Advanced SIMD registers.

**Table C3-26 Permitted combinations of parameters for VSTn (single n-element structure to one lane)**

<i>n</i>	<i>datatype</i>	<i>list as</i>	<i>align at</i>	<b>alignment</b>
1	8	{Dd[x]}	-	Standard only
	16	{Dd[x]}	@16	2-byte
	32	{Dd[x]}	@32	4-byte
2	8	{Dd[x], D(d+1)[x]}	@16	2-byte
	16	{Dd[x], D(d+1)[x]}	@32	4-byte

*as* Every register in the list must be in the range D0-D31.  
*at* *align* can be omitted. In this case, standard alignment rules apply.

**Table C3-26 Permitted combinations of parameters for VSTn (single n-element structure to one lane) (continued)**

<i>n</i>	<i>datatype</i>	<i>list as</i>	<i>align at</i>	<i>alignment</i>
		{Dd[x], D(d+2)[x]}	@32	4-byte
	32	{Dd[x], D(d+1)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x]}	@64	8-byte
3	8	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
	16 or 32	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
		{Dd[x], D(d+2)[x], D(d+4)[x]}	-	Standard only
4	8	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@32	4-byte
	16	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64	8-byte
	32	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64 or @128	8-byte or 16-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64 or @128	8-byte or 16-byte

**Related concepts**

[C3.3 Interleaving provided by load and store element and structure instructions](#) on page C3-449

[C3.4 Alignment restrictions in load and store element and structure instructions](#) on page C3-450

**Related reference**

[C1.9 Condition code suffixes](#) on page C1-142

## C3.134 VSTR

Extension register store.

### Syntax

`VSTR{cond}{.64} Dd, [Rn{, #offset}]`

where:

*cond*

is an optional condition code.

*Dd*

is the extension register to be saved.

*Rn*

is the general-purpose register holding the base address for the transfer.

*offset*

is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range -1020 to +1020. The value is added to the base address to form the address used for the transfer.

### Operation

The VSTR instruction saves the contents of an extension register to memory.

Two words are transferred.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

[C4.39 VSTR \(floating-point\) on page C4-640](#)

## C3.135 VSTR (post-increment and pre-decrement)

Pseudo-instruction that stores extension registers with post-increment and pre-decrement forms.

————— Note —————

There are also VLDR and VSTR instructions without post-increment and pre-decrement.

### Syntax

VSTR{cond}{.64} Dd, [Rn], #offset ; post-increment

VSTR{cond}{.64} Dd, [Rn, #-offset]! ; pre-decrement

where:

*cond*

is an optional condition code.

*Dd*

is the extension register to be saved.

*Rn*

is the general-purpose register holding the base address for the transfer.

*offset*

is a numeric expression that must evaluate to 8 at assembly time.

### Operation

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. This pseudo-instruction assembles to a VSTM instruction.

### Related reference

[C3.134 VSTR on page C3-585](#)

[C3.131 VSTM on page C3-580](#)

[C1.9 Condition code suffixes on page C1-142](#)

[C4.40 VSTR \(post-increment and pre-decrement, floating-point\) on page C4-641](#)

## C3.136 VSUB

Vector Subtract.

### Syntax

`VSUB{cond}.datatype {Qd}, Qn, Qm`

`VSUB{cond}.datatype {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I8, I16, I32, I64, or F32.

*Qd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

### Operation

VSUB subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.137 VSUBHN

Vector Subtract and Narrow, selecting High half.

### Syntax

`VSUBHN{cond}.datatype Dd, Qn, Qm`

where:

*cond*

is an optional condition code.

*datatype*

must be one of I16, I32, or I64.

*Dd, Qn, Qm*

are the destination vector, the first operand vector, and the second operand vector.

### Operation

VSUBHN subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, selects the most significant halves of the results, and places the final results in the destination doubleword vector. Results are truncated.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.138 VSUBL and VSUBW

Vector Subtract Long, Vector Subtract Wide.

### Syntax

`VSUBL{cond}.datatype Qd, Dn, Dm ; Long operation`

`VSUBW{cond}.datatype {Qd}, Qn, Dm ; Wide operation`

where:

*cond*

is an optional condition code.

*datatype*

must be one of S8, S16, S32, U8, U16, or U32.

*Qd, Dn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

*Qd, Qn, Dm*

are the destination vector, the first operand vector, and the second operand vector, for a wide operation.

### Operation

VSUBL subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in the destination quadword vector.

VSUBW subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in the destination quadword vector.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.139 VSWP

Vector Swap.

### Syntax

`VSWP{cond}{.datatype} Qd, Qm`

`VSWP{cond}{.datatype} Dd, Dm`

where:

*cond*

is an optional condition code.

*datatype*

is an optional datatype. The assembler ignores *datatype*.

*Qd, Qm*

specifies the vectors for a quadword operation.

*Dd, Dm*

specifies the vectors for a doubleword operation.

### Operation

VSWP exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.140 VTBL and VTBX

Vector Table Lookup, Vector Table Extension.

### Syntax

*Vop{cond}.8 Dd, List, Dm*

where:

*op*

must be either TBL or TBX.

*cond*

is an optional condition code.

*Dd*

specifies the destination vector.

*List*

Specifies the vectors containing the table. It must be one of:

- {*Dn*}.
- {*Dn, D(n+1)*}.
- {*Dn, D(n+1), D(n+2)*}.
- {*Dn, D(n+1), D(n+2), D(n+3)*}.
- {*Qn, Q(n+1)*}.

All the registers in *List* must be in the range D0-D31 or Q0-Q15 and must not wrap around the end of the register bank. For example {D31,D0,D1} is not permitted. If *List* contains Q registers, they disassemble to the equivalent D registers.

*Dm*

specifies the index vector.

### Operation

VTBL uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return zero.

VTBX works in the same way, except that indexes out of range leave the destination element unchanged.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.141 VTRN

Vector Transpose.

### Syntax

`VTRN{cond}.size Qd, Qm`

`VTRN{cond}.size Dd, Dm`

where:

*cond*

is an optional condition code.

*size*

must be one of 8, 16, or 32.

*Qd, Qm*

specifies the vectors, for a quadword operation.

*Dd, Dm*

specifies the vectors, for a doubleword operation.

### Operation

VTRN treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices. The following figures show examples of the operation of VTRN:

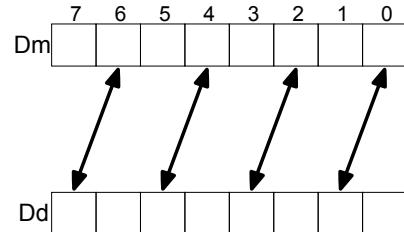


Figure C3-9 Operation of doubleword VTRN.8

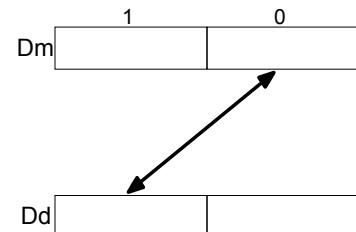


Figure C3-10 Operation of doubleword VTRN.32

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.142 VTST

Vector Test bits.

### Syntax

`VTST{cond}.size {Qd}, Qn, Qm`

`VTST{cond}.size {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*size*

must be one of 8, 16, or 32.

*Qd, Qn, Qm*

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

*Dd, Dn, Dm*

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

### Operation

VTST takes each element in a vector, and bitwise logical ANDs them with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C3.143 VUDOT (vector)

Dot Product vector form with unsigned integers.

### Syntax

VUDOT{q}.U8 Dd, Dn, Dm ; 64-bit SIMD vector

VUDOT{q}.U8 Qd, Qn, Qm ; A1 128-bit SIMD vector FP/SIMD registers (A32)

Where:

- q** Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.
- Dd** Is the 64-bit name of the SIMD and FP destination register.
- Dn** Is the 64-bit name of the first SIMD and FP source register.
- Dm** Is the 64-bit name of the second SIMD and FP source register.
- Qd** Is the 128-bit name of the SIMD and FP destination register.
- Qn** Is the 128-bit name of the first SIMD and FP source register.
- Qm** Is the 128-bit name of the second SIMD and FP source register.

### Architectures supported

Supported in Armv8.2 and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

### Usage

Dot Product vector form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

---

#### Note

---

ID\_ISAR6.DP indicates whether this instruction is supported in the T32 and A32 instruction sets.

---

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page C3-445

## C3.144 VUDOT (by element)

Dot Product index form with unsigned integers.

### Syntax

`VUDOT{q}.U8 Dd, Dn, Dm[index] ; 64-bit SIMD vector`

`VUDOT{q}.U8 Qd, Qn, Dm[index] ; A1 128-bit SIMD vector FP/SIMD registers (A32)`

Where:

- q*** Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page [C2-161](#).
- Dd*** Is the 64-bit name of the SIMD and FP destination register.
- Dn*** Is the 64-bit name of the first SIMD and FP source register.
- Dm*** Is the 64-bit name of the second SIMD and FP source register.
- index*** Is the element index in the range 0 to 1.
- Qd*** Is the 128-bit name of the SIMD and FP destination register.
- Qn*** Is the 128-bit name of the first SIMD and FP source register.

### Architectures supported

Supported in Armv8.2 and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

### Usage

Dot Product index form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

---

#### Note

---

ID\_ISAR6.DP indicates whether this instruction is supported in the T32 and A32 instruction sets.

---

### Related reference

[C3.1 Summary of Advanced SIMD instructions](#) on page [C3-445](#)

## C3.145 VUZP

Vector Unzip.

### Syntax

`VUZP{cond}.size Qd, Qm`

`VUZP{cond}.size Dd, Dm`

where:

`cond`

is an optional condition code.

`size`

must be one of 8, 16, or 32.

`Qd, Qm`

specifies the vectors, for a quadword operation.

`Dd, Dm`

specifies the vectors, for a doubleword operation.

### Note

The following are all the same instruction:

- `VZIP.32 Dd, Dm.`
- `VUZP.32 Dd, Dm.`
- `VTRN.32 Dd, Dm.`

The instruction is disassembled as `VTRN.32 Dd, Dm.`

### Operation

`VUZP` de-interleaves the elements of two vectors.

De-interleaving is the inverse process of interleaving.

**Table C3-27 Operation of doubleword VUZP.8**

	Register state before operation								Register state after operation							
Dd	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	B <sub>6</sub>	B <sub>4</sub>	B <sub>2</sub>	B <sub>0</sub>	A <sub>6</sub>	A <sub>4</sub>	A <sub>2</sub>	A <sub>0</sub>
Dm	B <sub>7</sub>	B <sub>6</sub>	B <sub>5</sub>	B <sub>4</sub>	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	B <sub>7</sub>	B <sub>5</sub>	B <sub>3</sub>	B <sub>1</sub>	A <sub>7</sub>	A <sub>5</sub>	A <sub>3</sub>	A <sub>1</sub>

**Table C3-28 Operation of quadword VUZP.32**

	Register state before operation				Register state after operation			
Qd	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	B <sub>2</sub>	B <sub>0</sub>	A <sub>2</sub>	A <sub>0</sub>
Qm	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	B <sub>3</sub>	B <sub>1</sub>	A <sub>3</sub>	A <sub>1</sub>

### Related concepts

[C3.3 Interleaving provided by load and store element and structure instructions](#) on page C3-449

### Related reference

[C3.141 VTRN](#) on page C3-592

[C1.9 Condition code suffixes](#) on page C1-142

## C3.146 VZIP

Vector Zip.

### Syntax

`VZIP{cond}.size Qd, Qm`

`VZIP{cond}.size Dd, Dm`

where:

*cond*

is an optional condition code.

*size*

must be one of 8, 16, or 32.

*Qd, Qm*

specifies the vectors, for a quadword operation.

*Dd, Dm*

specifies the vectors, for a doubleword operation.

### Note

The following are all the same instruction:

- `VZIP.32 Dd, Dm.`
- `VUZP.32 Dd, Dm.`
- `VTRN.32 Dd, Dm.`

The instruction is disassembled as `VTRN.32 Dd, Dm.`

### Operation

VZIP interleaves the elements of two vectors.

**Table C3-29 Operation of doubleword VZIP.8**

	Register state before operation								Register state after operation							
Dd	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	B <sub>3</sub>	A <sub>3</sub>	B <sub>2</sub>	A <sub>2</sub>	B <sub>1</sub>	A <sub>1</sub>	B <sub>0</sub>	A <sub>0</sub>
Dm	B <sub>7</sub>	B <sub>6</sub>	B <sub>5</sub>	B <sub>4</sub>	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	B <sub>7</sub>	A <sub>7</sub>	B <sub>6</sub>	A <sub>6</sub>	B <sub>5</sub>	A <sub>5</sub>	B <sub>4</sub>	A <sub>4</sub>

**Table C3-30 Operation of quadword VZIP.32**

	Register state before operation				Register state after operation			
Qd	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	B <sub>1</sub>	A <sub>1</sub>	B <sub>0</sub>	A <sub>0</sub>
Qm	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	B <sub>3</sub>	A <sub>3</sub>	B <sub>2</sub>	A <sub>2</sub>

### Related concepts

[C3.3 Interleaving provided by load and store element and structure instructions](#) on page C3-449

### Related reference

[C3.141 VTRN](#) on page C3-592

[C1.9 Condition code suffixes](#) on page C1-142



# Chapter C4

## Floating-point Instructions (32-bit)

Describes floating-point assembly language instructions.

It contains the following sections:

- [C4.1 Summary of floating-point instructions](#) on page C4-601.
- [C4.2 VABS \(floating-point\)](#) on page C4-603.
- [C4.3 VADD \(floating-point\)](#) on page C4-604.
- [C4.4 VCMP, VCMPE](#) on page C4-605.
- [C4.5 VCVT \(between single-precision and double-precision\)](#) on page C4-606.
- [C4.6 VCVT \(between floating-point and integer\)](#) on page C4-607.
- [C4.7 VCVT \(from floating-point to integer with directed rounding modes\)](#) on page C4-608.
- [C4.8 VCVT \(between floating-point and fixed-point\)](#) on page C4-609.
- [C4.9 VCVTB, VCVTT \(half-precision extension\)](#) on page C4-610.
- [C4.10 VCVTB, VCVTT \(between half-precision and double-precision\)](#) on page C4-611.
- [C4.11 VDIV](#) on page C4-612.
- [C4.12 VFMA, VFMS, VFNMA, VFNMS \(floating-point\)](#) on page C4-613.
- [C4.13 VJCVT](#) on page C4-614.
- [C4.14 VLDM \(floating-point\)](#) on page C4-615.
- [C4.15 VLDR \(floating-point\)](#) on page C4-616.
- [C4.16 VLDR \(post-increment and pre-decrement, floating-point\)](#) on page C4-617.
- [C4.17 VLLDM](#) on page C4-618.
- [C4.18 VLSTM](#) on page C4-619.
- [C4.19 VMAXNM, VMINNM \(floating-point\)](#) on page C4-620.
- [C4.20 VMLA \(floating-point\)](#) on page C4-621.
- [C4.21 VMLS \(floating-point\)](#) on page C4-622.
- [C4.22 VMOV \(floating-point\)](#) on page C4-623.

- C4.23 *VMOV* (*between one general-purpose register and single precision floating-point register*) on page C4-624.
- C4.24 *VMOV* (*between two general-purpose registers and one or two extension registers*) on page C4-625.
- C4.25 *VMOV* (*between a general-purpose register and half a double precision floating-point register*) on page C4-626.
- C4.26 *VMRS* (*floating-point*) on page C4-627.
- C4.27 *VMSR* (*floating-point*) on page C4-628.
- C4.28 *VMUL* (*floating-point*) on page C4-629.
- C4.29 *VNEG* (*floating-point*) on page C4-630.
- C4.30 *VNMLA* (*floating-point*) on page C4-631.
- C4.31 *VNMLS* (*floating-point*) on page C4-632.
- C4.32 *VNMUL* (*floating-point*) on page C4-633.
- C4.33 *VPOP* (*floating-point*) on page C4-634.
- C4.34 *VPUSH* (*floating-point*) on page C4-635.
- C4.35 *VRINT* (*floating-point*) on page C4-636.
- C4.36 *VSEL* on page C4-637.
- C4.37 *VSQRT* on page C4-638.
- C4.38 *VSTM* (*floating-point*) on page C4-639.
- C4.39 *VSTR* (*floating-point*) on page C4-640.
- C4.40 *VSTR* (*post-increment and pre-decrement, floating-point*) on page C4-641.
- C4.41 *VSUB* (*floating-point*) on page C4-642.

## C4.1 Summary of floating-point instructions

A summary of the floating-point instructions. Not all of these instructions are available in all floating-point versions.

The following table shows a summary of floating-point instructions that are not available in Advanced SIMD.

————— Note —————

Floating-point vector mode is not supported in Armv8. Use Advanced SIMD instructions for vector floating-point.

**Table C4-1 Summary of floating-point instructions**

Mnemonic	Brief description
VABS	Absolute value
VADD	Add
VCMP, VCMPE	Compare
VCVT	Convert between single-precision and double-precision
	Convert between floating-point and integer
	Convert between floating-point and fixed-point
	Convert floating-point to integer with directed rounding modes
VCVTB, VCVTT	Convert between half-precision and single-precision floating-point
	Convert between half-precision and double-precision
VDIV	Divide
VFMA, VFMS	Fused multiply accumulate, Fused multiply subtract
VFNMA, VFNMS	Fused multiply accumulate with negation, Fused multiply subtract with negation
VJCVT	Javascript Convert to signed fixed-point, rounding toward Zero
VLDM	Extension register load multiple
VLDR	Extension register load
VLLDM	Floating-point Lazy Load Multiple
VLSTM	Floating-point Lazy Store Multiple
VMAXNM, VMINNM	Maximum, Minimum, consistent with IEEE 754-2008
VMLA	Multiply accumulate
VMLS	Multiply subtract
VMOV	Insert floating-point immediate in single-precision or double-precision register, or copy one FP register into another FP register of the same width
VMRS	Transfer contents from a floating-point system register to a general-purpose register
VMSR	Transfer contents from a general-purpose register to a floating-point system register
VMUL	Multiply
VNEG	Negate

Table C4-1 Summary of floating-point instructions (continued)

Mnemonic	Brief description
VNMLA	Negated multiply accumulate
VNMLS	Negated multiply subtract
VNMUL	Negated multiply
VPOP	Extension register load multiple
VPUSH	Extension register store multiple
VRINT	Round to integer
VSEL	Select
VSQRT	Square Root
VSTM	Extension register store multiple
VSTR	Extension register store
VSUB	Subtract

## C4.2 VABS (floating-point)

Floating-point absolute value.

### Syntax

VABS{cond}.F32 *Sd, Sm*

VABS{cond}.F64 *Dd, Dm*

where:

*cond*

is an optional condition code.

*Sd, Sm*

are the single-precision registers for the result and operand.

*Dd, Dm*

are the double-precision registers for the result and operand.

### Operation

The VABS instruction takes the contents of *Sm* or *Dm*, clears the sign bit, and places the result in *Sd* or *Dd*. This gives the absolute value.

If the operand is a NaN, the sign bit is cleared, but no exception is produced.

### Floating-point exceptions

VABS instructions do not produce any exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.3 VADD (floating-point)

Floating-point add.

### Syntax

VADD{cond}.F32 {Sd}, Sn, Sm

VADD{cond}.F64 {Dd}, Dn, Dm

where:

*cond*

is an optional condition code.

*Sd, Sn, Sm*

are the single-precision registers for the result and operands.

*Dd, Dn, Dm*

are the double-precision registers for the result and operands.

### Operation

The VADD instruction adds the values in the operand registers and places the result in the destination register.

### Floating-point exceptions

The VADD instruction can produce Invalid Operation, Overflow, or Inexact exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.4 VCMP, VCMPE

Floating-point compare.

### Syntax

VCMP{E}{cond}.F32 *Sd*, *Sm*

VCMP{E}{cond}.F32 *Sd*, #0

VCMP{E}{cond}.F64 *Dd*, *Dm*

VCMP{E}{cond}.F64 *Dd*, #0

where:

**E**

if present, indicates that the instruction raises an Invalid Operation exception if either operand is a quiet or signaling NaN. Otherwise, it raises the exception only if either operand is a signaling NaN.

*cond*

is an optional condition code.

*Sd*, *Sm*

are the single-precision registers holding the operands.

*Dd*, *Dm*

are the double-precision registers holding the operands.

### Operation

The VCMP{E} instruction subtracts the value in the second operand register (or 0 if the second operand is #0) from the value in the first operand register, and sets the VFP condition flags based on the result.

### Floating-point exceptions

VCMP{E} instructions can produce Invalid Operation exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.5 VCVT (between single-precision and double-precision)

Convert between single-precision and double-precision numbers.

### Syntax

`VCVT{cond}.F64.F32 Dd, Sm`

`VCVT{cond}.F32.F64 Sd, Dm`

where:

*cond*

is an optional condition code.

*Dd*

is a double-precision register for the result.

*Sm*

is a single-precision register holding the operand.

*Sd*

is a single-precision register for the result.

*Dm*

is a double-precision register holding the operand.

### Operation

These instructions convert the single-precision value in *Sm* to double-precision, placing the result in *Dd*, or the double-precision value in *Dm* to single-precision, placing the result in *Sd*.

### Floating-point exceptions

These instructions can produce Invalid Operation, Input Denormal, Overflow, Underflow, or Inexact exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.6 VCVT (between floating-point and integer)

Convert between floating-point numbers and integers.

### Syntax

`VCVT{R}{cond}.type.F64 Sd, Dm`

`VCVT{R}{cond}.type.F32 Sd, Sm`

`VCVT{cond}.F64.type Dd, Sm`

`VCVT{cond}.F32.type Sd, Sm`

where:

`R`

makes the operation use the rounding mode specified by the FPSCR. Otherwise, the operation rounds towards zero.

`cond`

is an optional condition code.

`type`

can be either `U32` (unsigned 32-bit integer) or `S32` (signed 32-bit integer).

`Sd`

is a single-precision register for the result.

`Dd`

is a double-precision register for the result.

`Sm`

is a single-precision register holding the operand.

`Dm`

is a double-precision register holding the operand.

### Operation

The first two forms of this instruction convert from floating-point to integer.

The third and fourth forms convert from integer to floating-point.

### Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.7 VCVT (from floating-point to integer with directed rounding modes)

Convert from floating-point to signed or unsigned integer with directed rounding modes.

————— Note —————

This instruction is supported only in Armv8.

### Syntax

*VCVTmode.S32.F64 Sd, Dm*

*VCVTmode.S32.F32 Sd, Sm*

*VCVTmode.U32.F64 Sd, Dm*

*VCVTmode.U32.F32 Sd, Sm*

where:

*mode*

must be one of:

**A**

meaning round to nearest, ties away from zero

**N**

meaning round to nearest, ties to even

**P**

meaning round towards plus infinity

**M**

meaning round towards minus infinity.

*Sd, Sm*

specifies the single-precision registers for the operand and result.

*Sd, Dm*

specifies a single-precision register for the result and double-precision register holding the operand.

### Notes

You cannot use VCVT with a directed rounding mode inside an IT block.

### Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.

## C4.8 VCVT (between floating-point and fixed-point)

Convert between floating-point and fixed-point numbers.

### Syntax

```
VCVT{cond}.type.F64 Dd, Dd, #fbits  
VCVT{cond}.type.F32 Sd, Sd, #fbits  
VCVT{cond}.F64.type Dd, Dd, #fbits  
VCVT{cond}.F32.type Sd, Sd, #fbits
```

where:

*cond*

is an optional condition code.

*type*

can be any one of:

**S16**

16-bit signed fixed-point number.

**U16**

16-bit unsigned fixed-point number.

**S32**

32-bit signed fixed-point number.

**U32**

32-bit unsigned fixed-point number.

*Sd*

is a single-precision register for the operand and result.

*Dd*

is a double-precision register for the operand and result.

*fbits*

is the number of fraction bits in the fixed-point number, in the range 0-16 if *type* is S16 or U16, or in the range 1-32 if *type* is S32 or U32.

### Operation

The first two forms of this instruction convert from floating-point to fixed-point.

The third and fourth forms convert from fixed-point to floating-point.

In all cases the fixed-point number is contained in the least significant 16 or 32 bits of the register.

### Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.9 VCVTB, VCVTT (half-precision extension)

Convert between half-precision and single-precision floating-point numbers.

### Syntax

`VCVTB{cond}.type Sd, Sm`

`VCVTT{cond}.type Sd, Sm`

where:

*cond*

is an optional condition code.

*type*

can be any one of:

**F32.F16**

Convert from half-precision to single-precision.

**F16.F32**

Convert from single-precision to half-precision.

*Sd*

is a single word register for the result.

*Sm*

is a single word register for the operand.

### Operation

VCVTB uses the bottom half (bits[15:0]) of the single word register to obtain or store the half-precision value

VCVTT uses the top half (bits[31:16]) of the single word register to obtain or store the half-precision value.

### Architectures

The instructions are only available in VFPv3 systems with the half-precision extension, and VFPv4.

### Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.10 VCVTB, VCVTT (between half-precision and double-precision)

These instructions convert between half-precision and double-precision floating-point numbers.

The conversion can be done in either of the following ways:

- From half-precision floating-point to double-precision floating-point (F64.F16).
- From double-precision floating-point to half-precision floating-point (F16.F64).

VCVTB uses the bottom half (bits[15:0]) of the single word register to obtain or store the half-precision value.

VCVTT uses the top half (bits[31:16]) of the single word register to obtain or store the half-precision value.

————— **Note** —————

These instructions are supported only in Armv8.

### Syntax

`VCVTB{cond}.F64.F16 Dd, Sm`

`VCVTB{cond}.F16.F64 Sd, Dm`

`VCVTT{cond}.F64.F16 Dd, Sm`

`VCVTT{cond}.F16.F64 Sd, Dm`

where:

*cond*

is an optional condition code.

*Dd*

is a double-precision register for the result.

*Sm*

is a single word register holding the operand.

*Sd*

is a single word register for the result.

*Dm*

is a double-precision register holding the operand.

### Usage

These instructions convert the half-precision value in *Sm* to double-precision and place the result in *Dd*, or the double-precision value in *Dm* to half-precision and place the result in *Sd*.

### Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

## C4.11 VDIV

Floating-point divide.

### Syntax

`VDIV{cond}.F32 {Sd}, Sn, Sm`

`VDIV{cond}.F64 {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*Sd, Sn, Sm*

are the single-precision registers for the result and operands.

*Dd, Dn, Dm*

are the double-precision registers for the result and operands.

### Operation

The VDIV instruction divides the value in the first operand register by the value in the second operand register, and places the result in the destination register.

### Floating-point exceptions

VDIV operations can produce Division by Zero, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.12 VFMA, VFMS, VFNMA, VFNMS (floating-point)

Fused floating-point multiply accumulate and fused floating-point multiply subtract, with optional negation.

### Syntax

$\text{VF}\{\text{N}\}\text{op}\{\text{cond}\}.\text{F64 } \{Dd\}, Dn, Dm$

$\text{VF}\{\text{N}\}\text{op}\{\text{cond}\}.\text{F32 } \{Sd\}, Sn, Sm$

where:

$\text{op}$

is one of MA or MS.

$\text{N}$

negates the final result.

$\text{cond}$

is an optional condition code.

$Sd, Sn, Sm$

are the single-precision registers for the result and operands.

$Dd, Dn, Dm$

are the double-precision registers for the result and operands.

### Operation

VFMA multiplies the values in the operand registers, adds the value in the destination register, and places the final result in the destination register. The result of the multiply is not rounded before the accumulation.

VFMS multiplies the values in the operand registers, subtracts the product from the value in the destination register, and places the final result in the destination register. The result of the multiply is not rounded before the subtraction.

In each case, the final result is negated if the N option is used.

### Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

### Related reference

[C4.28 VMUL \(floating-point\) on page C4-629](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C4.13 VJCVT

Javascript Convert to signed fixed-point, rounding toward Zero.

### Syntax

`VJCVT{q}.S32.F64 Sd, Dm ; A1 FP/SIMD registers (A32)`

`VJCVT{q}.S32.F64 Sd, Dm ; T1 FP/SIMD registers (T32)`

Where:

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**Sd**

Is the 32-bit name of the SIMD and FP destination register.

**Dm**

Is the 64-bit name of the SIMD and FP source register.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Javascript Convert to signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD and FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and write the result to the general-purpose destination register. If the result is too large to be held as a 32-bit signed integer, then the result is the integer modulo  $2^{32}$ , as held in a 32-bit signed integer.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[C4.1 Summary of floating-point instructions](#) on page C4-601

## C4.14 VLDM (floating-point)

Extension register load multiple.

### Syntax

`VLDMmode{cond} Rn{!}, Registers`

where:

*mode*

must be one of:

**IA**

meaning Increment address After each transfer. IA is the default, and can be omitted.

**DB**

meaning Decrement address Before each transfer.

**EA**

meaning Empty Ascending stack operation. This is the same as DB for loads.

**FD**

meaning Full Descending stack operation. This is the same as IA for loads.

*cond*

is an optional condition code.

*Rn*

is the general-purpose register holding the base address for the transfer.

!

is optional. ! specifies that the updated base address must be written back to *Rn*. If ! is not specified, *mode* must be IA.

*Registers*

is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify S or D registers, but they must not be mixed. The number of registers must not exceed 16 D registers.

---

### Note

---

VPOP *Registers* is equivalent to VLDM sp!, *Registers*.

You can use either form of this instruction. They both disassemble to VPOP.

---

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.15 VLDR (floating-point)

Extension register load.

### Syntax

`VLDR{cond}{.size} Fd, [Rn{, #offset}]`

`VLDR{cond}{.size} Fd, Label`

where:

*cond*

is an optional condition code.

*size*

is an optional data size specifier. Must be 32 if *Fd* is an S register, or 64 otherwise.

*Fd*

is the extension register to be loaded, and can be either a D or S register.

*Rn*

is the general-purpose register holding the base address for the transfer.

*offset*

is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range -1020 to +1020. The value is added to the base address to form the address used for the transfer.

*Label*

is a PC-relative expression.

*Label* must be aligned on a word boundary within ±1KB of the current instruction.

### Operation

The VLDR instruction loads an extension register from memory.

One word is transferred if *Fd* is an S register. Two words are transferred otherwise.

There is also a VLDR pseudo-instruction.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.16 VLDR (post-increment and pre-decrement, floating-point)

Pseudo-instruction that loads extension registers, with post-increment and pre-decrement forms.

————— Note ————

There are also VLDR and VSTR instructions without post-increment and pre-decrement.

### Syntax

`VLDR{cond}{.size} Fd, [Rn], #offset ; post-increment`

`VLDR{cond}{.size} Fd, [Rn, #-offset]! ; pre-decrement`

where:

*cond*

is an optional condition code.

*size*

is an optional data size specifier. Must be 32 if *Fd* is an S register, or 64 if *Fd* is a D register.

*Fd*

is the extension register to load. It can be either a double precision (*Dd*) or a single precision (*Sd*) register.

*Rn*

is the general-purpose register holding the base address for the transfer.

*offset*

is a numeric expression that must evaluate to a numeric value at assembly time. The value must be 4 if *Fd* is an S register, or 8 if *Fd* is a D register.

### Operation

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. This pseudo-instruction assembles to a VLDM instruction.

### Related reference

[C4.14 VLDM \(floating-point\) on page C4-615](#)

[C4.15 VLDR \(floating-point\) on page C4-616](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C4.17 VLLDM

Floating-point Lazy Load Multiple.

### Syntax

VLLDM{c}{q} Rn

Where:

**c**

Is an optional condition code. See [Chapter C1 Condition Codes](#) on page C1-133.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**Rn**

Is the general-purpose base register.

### Architectures supported

Supported in Armv8-M Main extension only.

### Usage

Floating-point Lazy Load Multiple restores the contents of the Secure floating-point registers that were protected by a VLSTM instruction, and marks the floating-point context as active.

If the lazy state preservation set up by a previous VLSTM instruction is active (FPCCR.LSPACT == 1), this instruction deactivates lazy state preservation and enables access to the Secure floating-point registers.

If lazy state preservation is inactive (FPCCR.LSPACT == 0), either because lazy state preservation was not enabled (FPCCR.LSPEN == 0) or because a floating-point instruction caused the Secure floating-point register contents to be stored to memory, this instruction loads the stored Secure floating-point register contents back into the floating-point registers.

If Secure floating-point is not in use (CONTROL\_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point Extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

### Related reference

[C4.1 Summary of floating-point instructions](#) on page C4-601

## C4.18 VLSTM

Floating-point Lazy Store Multiple.

### Syntax

`VLSTM{c}{q} Rn`

Where:

**c**

Is an optional condition code. See [Chapter C1 Condition Codes](#) on page C1-133.

**q**

Is an optional instruction width specifier. See [C2.2 Instruction width specifiers](#) on page C2-161.

**Rn**

Is the general-purpose base register.

### Architectures supported

Supported in Armv8-M Main extension only.

### Usage

Floating-point Lazy Store Multiple stores the contents of Secure floating-point registers to a prepared stack frame, and clears the Secure floating-point registers.

If floating-point lazy preservation is enabled (FPCCR.LSPEN == 1), then the next time a floating-point instruction other than VLSTM or VLLDM is executed:

- The contents of Secure floating-point registers are stored to memory.
- The Secure floating-point registers are cleared.

If Secure floating-point is not in use (CONTROL\_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

### Related reference

[C4.1 Summary of floating-point instructions](#) on page C4-601

## C4.19 VMAXNM, VMINNM (floating-point)

Vector Minimum, Vector Maximum.

————— Note ————

These instructions are supported only in Armv8.

### Syntax

*Vop.F32 Sd, Sn, Sm*

*Vop.F64 Dd, Dn, Dm*

where:

*op*

must be either MAXNM or MINNM.

*Sd, Sn, Sm*

are the single-precision destination register, first operand register, and second operand register.

*Dd, Dn, Dm*

are the double-precision destination register, first operand register, and second operand register.

### Operation

VMAXNM compares the values in the operand registers, and copies the larger value into the destination operand register.

VMINNM compares the values in the operand registers, and copies the smaller value into the destination operand register.

If one of the values being compared is a number and the other value is NaN, the number is copied into the destination operand register. This is consistent with the IEEE 754-2008 standard.

### Notes

You cannot use VMAXNM or VMINNM inside an IT block.

### Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

## C4.20 VMLA (floating-point)

Floating-point multiply accumulate.

### Syntax

VMLA{cond}.F32 *Sd*, *Sn*, *Sm*

VMLA{cond}.F64 *Dd*, *Dn*, *Dm*

where:

*cond*

is an optional condition code.

*Sd*, *Sn*, *Sm*

are the single-precision registers for the result and operands.

*Dd*, *Dn*, *Dm*

are the double-precision registers for the result and operands.

### Operation

The VMLA instruction multiplies the values in the operand registers, adds the value in the destination register, and places the final result in the destination register.

### Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C4.21 VMLS (floating-point)

Floating-point multiply subtract.

### Syntax

`VMLS{cond}.F32 Sd, Sn, Sm`

`VMLS{cond}.F64 Dd, Dn, Dm`

where:

*cond*

is an optional condition code.

*Sd, Sn, Sm*

are the single-precision registers for the result and operands.

*Dd, Dn, Dm*

are the double-precision registers for the result and operands.

### Operation

The VMLS instruction multiplies the values in the operand registers, subtracts the result from the value in the destination register, and places the final result in the destination register.

### Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.22 VMOV (floating-point)

Insert a floating-point immediate value into a single-precision or double-precision register, or copy one register into another register. This instruction is always scalar.

### Syntax

VMOV{cond}.F32 Sd, #imm

VMOV{cond}.F64 Dd, #imm

VMOV{cond}.F32 Sd, Sm

VMOV{cond}.F64 Dd, Dm

where:

*cond*

is an optional condition code.

*Sd*

is the single-precision destination register.

*Dd*

is the double-precision destination register.

*imm*

is the floating-point immediate value.

*Sm*

is the single-precision source register.

*Dm*

is the double-precision source register.

### Immediate values

Any number that can be expressed as  $\pm n * 2^{-r}$ , where  $n$  and  $r$  are integers,  $16 \leq n \leq 31$ ,  $0 \leq r \leq 7$ .

### Architectures

The instructions that copy immediate constants are available in VFPv3 and above.

The instructions that copy from registers are available in all VFP systems.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.23 VMOV (between one general-purpose register and single precision floating-point register)

Transfer contents between a single-precision floating-point register and a general-purpose register.

### Syntax

`VMOV{cond} Rd, Sn`

`VMOV{cond} Sn, Rd`

where:

*cond*

is an optional condition code.

*Sn*

is the floating-point single-precision register.

*Rd*

is the general-purpose register. *Rd* must not be PC.

### Operation

`VMOV Rd, Sn` transfers the contents of *Sn* into *Rd*.

`VMOV Sn, Rd` transfers the contents of *Rd* into *Sn*.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C4.24 VMOV (between two general-purpose registers and one or two extension registers)

Transfer contents between two general-purpose registers and either one 64-bit register or two consecutive 32-bit registers.

### Syntax

`VMOV{cond} Dm, Rd, Rn`

`VMOV{cond} Rd, Rn, Dm`

`VMOV{cond} Sm, Sm1, Rd, Rn`

`VMOV{cond} Rd, Rn, Sm, Sm1`

where:

*cond*

is an optional condition code.

*Dm*

is a 64-bit extension register.

*Sm*

is a VFP 32-bit register.

*Sm1*

is the next consecutive VFP 32-bit register after *Sm*.

*Rd, Rn*

are the general-purpose registers. *Rd* and *Rn* must not be PC.

### Operation

`VMOV Dm, Rd, Rn` transfers the contents of *Rd* into the low half of *Dm*, and the contents of *Rn* into the high half of *Dm*.

`VMOV Rd, Rn, Dm` transfers the contents of the low half of *Dm* into *Rd*, and the contents of the high half of *Dm* into *Rn*.

`VMOV Rd, Rn, Sm, Sm1` transfers the contents of *Sm* into *Rd*, and the contents of *Sm1* into *Rn*.

`VMOV Sm, Sm1, Rd, Rn` transfers the contents of *Rd* into *Sm*, and the contents of *Rn* into *Sm1*.

### Architectures

The instructions are available in VFPv2 and above.

#### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.25 VMOV (between a general-purpose register and half a double precision floating-point register)

Transfer contents between a general-purpose register and half a double precision floating-point register.

### Syntax

`VMOV{cond}{.size} Dn[x], Rd`

`VMOV{cond}{.size} Rd, Dn[x]`

where:

*cond*

is an optional condition code.

*size*

the data size. Must be either 32 or omitted. If omitted, *size* is 32.

*Dn[x]*

is the upper or lower half of a double precision floating-point register.

*Rd*

is the general-purpose register. *Rd* must not be PC.

### Operation

`VMOV Dn[x], Rd` transfers the contents of *Rd* into *Dn[x]*.

`VMOV Rd, Dn[x]` transfers the contents of *Dn[x]* into *Rd*.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.26 VMRS (floating-point)

Transfer contents from an floating-point system register to a general-purpose register.

### Syntax

VMRS{cond} Rd, extsysreg

where:

*cond*

is an optional condition code.

*extsysreg*

is the floating-point system register, usually FPSCR, FPSID, or FPEXC.

*Rd*

is the general-purpose register. *Rd* must not be PC.

It can be APSR\_nzcv, if *extsysreg* is FPSCR. In this case, the floating-point status flags are transferred into the corresponding flags in the special-purpose APSR.

### Usage

The VMRS instruction transfers the contents of *extsysreg* into *Rd*.

---

#### Note

---

The instruction stalls the processor until all current floating-point operations complete.

---

### Examples

```
VMRS    r2, FPCID
VMRS    APSR_nzcv, FPSCR      ; transfer FP status register to the
                                ; special-purpose APSR
```

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C4.27 VMSR (floating-point)

Transfer contents of a general-purpose register to a floating-point system register.

### Syntax

VMSR{cond} extsysreg, Rd

where:

*cond*

is an optional condition code.

*extsysreg*

is the floating-point system register, usually FPSCR, FPSID, or FPEXC.

*Rd*

is the general-purpose register. *Rd* must not be PC.

It can be APSR\_nzcv, if *extsysreg* is FPSCR. In this case, the floating-point status flags are transferred into the corresponding flags in the special-purpose APSR.

### Usage

The VMSR instruction transfers the contents of *Rd* into *extsysreg*.

---

#### Note

---

The instruction stalls the processor until all current floating-point operations complete.

---

### Example

VMSR      FPSCR, r4

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.28 VMUL (floating-point)

Floating-point multiply.

### Syntax

`VMUL{cond}.F32 {Sd,} Sn, Sm`

`VMUL{cond}.F64 {Dd,} Dn, Dm`

where:

*cond*

is an optional condition code.

*Sd, Sn, Sm*

are the single-precision registers for the result and operands.

*Dd, Dn, Dm*

are the double-precision registers for the result and operands.

### Operation

The VMUL operation multiplies the values in the operand registers and places the result in the destination register.

### Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.29 VNEG (floating-point)

Floating-point negate.

### Syntax

VNEG{cond}.F32 *Sd, Sm*

VNEG{cond}.F64 *Dd, Dm*

where:

*cond*

is an optional condition code.

*Sd, Sm*

are the single-precision registers for the result and operand.

*Dd, Dm*

are the double-precision registers for the result and operand.

### Operation

The VNEG instruction takes the contents of *Sm* or *Dm*, changes the sign bit, and places the result in *Sd* or *Dd*. This gives the negation of the value.

If the operand is a NaN, the sign bit is changed, but no exception is produced.

### Floating-point exceptions

VNEG instructions do not produce any exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.30 VNMLA (floating-point)

Floating-point multiply accumulate with negation.

### Syntax

`VNMLA{cond}.F32 Sd, Sn, Sm`

`VNMLA{cond}.F64 Dd, Dn, Dm`

where:

*cond*

is an optional condition code.

*Sd, Sn, Sm*

are the single-precision registers for the result and operands.

*Dd, Dn, Dm*

are the double-precision registers for the result and operands.

### Operation

The VNMLA instruction multiplies the values in the operand registers, adds the value to the destination register, and places the negated final result in the destination register.

### Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C4.31 VNMLS (floating-point)

Floating-point multiply subtract with negation.

### Syntax

`VNMLS{cond}.F32 Sd, Sn, Sm`

`VNMLS{cond}.F64 Dd, Dn, Dm`

where:

*cond*

is an optional condition code.

*Sd, Sn, Sm*

are the single-precision registers for the result and operands.

*Dd, Dn, Dm*

are the double-precision registers for the result and operands.

### Operation

The VNMLS instruction multiplies the values in the operand registers, subtracts the result from the value in the destination register, and places the negated final result in the destination register.

### Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

### Related reference

[C1.9 Condition code suffixes](#) on page C1-142

## C4.32 VNMUL (floating-point)

Floating-point multiply with negation.

### Syntax

VNMUL{cond}.F32 {Sd,} Sn, Sm

VNMUL{cond}.F64 {Dd,} Dn, Dm

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

### Operation

The VNMUL instruction multiplies the values in the operand registers and places the negated result in the destination register.

### Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.33 VPOP (floating-point)

Pop extension registers from the stack.

### Syntax

`VPOP{cond} Registers`

where:

*cond*

is an optional condition code.

*Registers*

is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify S or D registers, but they must not be mixed. The number of registers must not exceed 16 D registers.

---

### Note

---

`VPOP Registers` is equivalent to `VLDM sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPOP`.

---

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

[C4.34 VPUSH \(floating-point\) on page C4-635](#)

## C4.34 VPUSH (floating-point)

Push extension registers onto the stack.

### Syntax

`VPUSH{cond} Registers`

where:

*cond*

is an optional condition code.

*Registers*

is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify S or D registers, but they must not be mixed. The number of registers must not exceed 16 D registers.

---

### Note

---

`VPUSH Registers` is equivalent to `VSTMDB sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPUSH`.

---

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

[C4.33 VPOP \(floating-point\) on page C4-634](#)

## C4.35 VRINT (floating-point)

Rounds a floating-point number to integer and places the result in the destination register. The resulting integer is represented in floating-point format.

— Note —

This instruction is supported only in Armv8.

### Syntax

`VRINTmode{cond}.F64.F64 Dd, Dm`

`VRINTmode{cond}.F32.F32 Sd, Sm`

where:

*mode*

must be one of:

**Z**

meaning round towards zero.

**R**

meaning use the rounding mode specified in the FPSCR.

**X**

meaning use the rounding mode specified in the FPSCR, generating an Inexact exception if the result is not exact.

**A**

meaning round to nearest, ties away from zero.

**N**

meaning round to nearest, ties to even.

**P**

meaning round towards plus infinity.

**M**

meaning round towards minus infinity.

*cond*

is an optional condition code. This can only be used when *mode* is Z, R or X.

*Sd, Sm*

specifies the destination and operand registers, for a word operation.

*Dd, Dm*

specifies the destination and operand registers, for a doubleword operation.

### Notes

You cannot use VRINT with a rounding mode of A, N, P or M inside an IT block.

### Floating-point exceptions

These instructions cannot produce any exceptions, except VRINTX which can generate an Inexact exception.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.36 VSEL

Floating-point select.

— Note —

This instruction is supported only in Armv8.

### Syntax

`VSELcond.F32 Sd, Sn, Sm`

`VSELcond.F64 Dd, Dn, Dm`

where:

*cond*

must be one of GE, GT, EQ, VS.

*Sd, Sn, Sm*

are the single-precision registers for the result and operands.

*Dd, Dn, Dm*

are the double-precision registers for the result and operands.

### Usage

The VSEL instruction compares the values in the operand registers. If the condition is true, it copies the value in the first operand register into the destination operand register. Otherwise, it copies the value in the second operand register.

You cannot use VSEL inside an IT block.

### Floating-point exceptions

VSEL instructions cannot produce any exceptions.

#### Related reference

[C1.11 Comparison of condition code meanings in integer and floating-point code](#) on page C1-144

[C1.9 Condition code suffixes](#) on page C1-142

## C4.37 VSQRT

Floating-point square root.

### Syntax

`VSQRT{cond}.F32 Sd, Sm`

`VSQRT{cond}.F64 Dd, Dm`

where:

*cond*

is an optional condition code.

*Sd, Sm*

are the single-precision registers for the result and operand.

*Dd, Dm*

are the double-precision registers for the result and operand.

### Operation

The VSQRT instruction takes the square root of the contents of *Sm* or *Dm*, and places the result in *Sd* or *Dd*.

### Floating-point exceptions

VSQRT instructions can produce Invalid Operation or Inexact exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.38 VSTM (floating-point)

Extension register store multiple.

### Syntax

`VSTMmode{cond} Rn{!}, Registers`

where:

*mode*

must be one of:

**IA**

meaning Increment address After each transfer. IA is the default, and can be omitted.

**DB**

meaning Decrement address Before each transfer.

**EA**

meaning Empty Ascending stack operation. This is the same as IA for stores.

**FD**

meaning Full Descending stack operation. This is the same as DB for stores.

*cond*

is an optional condition code.

*Rn*

is the general-purpose register holding the base address for the transfer.

!

is optional. ! specifies that the updated base address must be written back to *Rn*. If ! is not specified, *mode* must be IA.

*Registers*

is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify S or D registers, but they must not be mixed. The number of registers must not exceed 16 D registers.

---

### Note

---

`VPUSH Registers` is equivalent to `VSTMDB sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPUSH`.

---

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.39 VSTR (floating-point)

Extension register store.

### Syntax

`VSTR{cond}{.size} Fd, [Rn{, #offset}]`

where:

*cond*

is an optional condition code.

*size*

is an optional data size specifier. Must be 32 if *Fd* is an S register, or 64 otherwise.

*Fd*

is the extension register to be saved. It can be either a D or S register.

*Rn*

is the general-purpose register holding the base address for the transfer.

*offset*

is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range -1020 to +1020. The value is added to the base address to form the address used for the transfer.

### Operation

The VSTR instruction saves the contents of an extension register to memory.

One word is transferred if *Fd* is an S register. Two words are transferred otherwise.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

## C4.40 VSTR (post-increment and pre-decrement, floating-point)

Pseudo-instruction that stores extension registers with post-increment and pre-decrement forms.

————— Note —————

There are also VLDR and VSTR instructions without post-increment and pre-decrement.

### Syntax

`VSTR{cond}{.size} Fd, [Rn], #offset ; post-increment`

`VSTR{cond}{.size} Fd, [Rn, #-offset]! ; pre-decrement`

where:

*cond*

is an optional condition code.

*size*

is an optional data size specifier. Must be 32 if *Fd* is an S register, or 64 if *Fd* is a D register.

*Fd*

is the extension register to be saved. It can be either a double precision (*Dd*) or a single precision (*Sd*) register.

*Rn*

is the general-purpose register holding the base address for the transfer.

*offset*

is a numeric expression that must evaluate to a numeric value at assembly time. The value must be 4 if *Fd* is an S register, or 8 if *Fd* is a D register.

### Operation

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. This pseudo-instruction assembles to a VSTM instruction.

### Related reference

[C4.39 VSTR \(floating-point\) on page C4-640](#)

[C4.38 VSTM \(floating-point\) on page C4-639](#)

[C1.9 Condition code suffixes on page C1-142](#)

## C4.41 VSUB (floating-point)

Floating-point subtract.

### Syntax

`VSUB{cond}.F32 {Sd}, Sn, Sm`

`VSUB{cond}.F64 {Dd}, Dn, Dm`

where:

*cond*

is an optional condition code.

*Sd, Sn, Sm*

are the single-precision registers for the result and operands.

*Dd, Dn, Dm*

are the double-precision registers for the result and operands.

### Operation

The VSUB instruction subtracts the value in the second operand register from the value in the first operand register, and places the result in the destination register.

### Floating-point exceptions

The VSUB instruction can produce Invalid Operation, Overflow, or Inexact exceptions.

### Related reference

[C1.9 Condition code suffixes on page C1-142](#)

# Chapter C5

## A32/T32 Cryptographic Algorithms

Lists the algorithms that A32 and T32 SIMD instructions support.

It contains the following section:

- [\*C5.1 A32/T32 Cryptographic instructions\* on page C5-644.](#)

## C5.1 A32/T32 Cryptographic instructions

A set of A32 and T32 cryptographic instructions is available in the Armv8 architecture.

These instructions use the 128-bit Advanced SIMD registers and support the acceleration of the following cryptographic and hash algorithms:

- AES.
- SHA1.
- SHA256.

### Summary of A32/T32 cryptographic instructions

The following table lists the A32/T32 cryptographic instructions that are supported:

Table C5-1 Summary of A32/T32 cryptographic instructions

Mnemonic	Brief description
AESD	AES single round decryption
AESE	AES single round encryption
AESIMC	AES inverse mix columns
AESMC	AES mix columns
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
SHA256H2	SHA256 hash update part 2
SHA256H	SHA256 hash update part 1
SHA256SU0	SHA256 schedule update 0
SHA256SU1	SHA256 schedule update 1

## **Part D**

# **A64 Instruction Set Reference**



# Chapter D1

## Condition Codes

Describes condition codes and conditional execution of A64 code.

It contains the following sections:

- [\*D1.1 Conditional execution in A64 code\* on page D1-648.](#)
- [\*D1.2 Condition flags\* on page D1-649.](#)
- [\*D1.3 Updates to the condition flags in A64 code\* on page D1-650.](#)
- [\*D1.4 Floating-point instructions that update the condition flags\* on page D1-651.](#)
- [\*D1.5 Carry flag\* on page D1-652.](#)
- [\*D1.6 Overflow flag\* on page D1-653.](#)
- [\*D1.7 Condition code suffixes\* on page D1-654.](#)
- [\*D1.8 Condition code suffixes and related flags\* on page D1-655.](#)
- [\*D1.9 Optimization for execution speed\* on page D1-656.](#)

## D1.1 Conditional execution in A64 code

In the A64 instruction set, there are a few instructions that are truly conditional. Truly conditional means that when the condition is false, the instruction advances the program counter but has no other effect.

The conditional branch, `B.cond` is a truly conditional instruction. The condition code is appended to the instruction with a '!' delimiter, for example `B.EQ`.

There are other truly conditional branch instructions that execute depending on the value of the Zero condition flag. You cannot append any condition code suffix to them. These instructions are:

- `CBNZ`.
- `CBZ`.
- `TBNZ`.
- `TBZ`.

There are a few A64 instructions that are unconditionally executed but use the condition code as a source operand. These instructions always execute but the operation depends on the value of the condition code. These instructions can be categorized as:

- Conditional data processing instructions, for example `CSEL`.
- Conditional comparison instructions, `CCMN` and `CCMP`.

In these instructions, you specify the condition code in the final operand position, for example `CSEL Wd,Wm,Wn,NE`.

## D1.2 Condition flags

The N, Z, C, and V condition flags are held in the APSR.

The condition flags are held in the APSR. They are set or cleared as follows:

**N**

Set to 1 when the result of the operation is negative, cleared to 0 otherwise.

**Z**

Set to 1 when the result of the operation is zero, cleared to 0 otherwise.

**C**

Set to 1 when the operation results in a carry, or when a subtraction results in no borrow, cleared to 0 otherwise.

**V**

Set to 1 when the operation causes overflow, cleared to 0 otherwise.

C is set in one of the following ways:

- For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-addition/subtractions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.

Overflow occurs if the result of a signed add, subtract, or compare is greater than or equal to  $2^{31}$ , or less than  $-2^{31}$ .

### **Related reference**

[C1.5 Updates to the condition flags in A32/T32 code on page C1-138](#)

[C1.10 Condition code suffixes and related flags on page C1-143](#)

[D1.3 Updates to the condition flags in A64 code on page D1-650](#)

[D1.8 Condition code suffixes and related flags on page D1-655](#)

## D1.3 Updates to the condition flags in A64 code

In AArch64 state, the N, Z, C, and V condition flags are held in the NZCV system register, which is part of the process state. You can access the flags using the MSR and MRS instructions.

— Note —

An instruction updates the condition flags only if the S suffix is specified, except the instructions CMP, CMN, CCMP, CCMN, and TST, which always update the condition flags. The instruction also determines which flags get updated. If a conditional instruction does not execute, it does not affect the flags.

### Example

This example shows the read-modify-write procedure to change some of the condition flags in A64 code.

```
MRS  x1, NZCV          ; copy N, Z, C, and V flags into general-purpose x1
MOV  x2, #0x30000000
BIC  x1,x1,x2          ; clears the C and V flags (bits 29,28)
ORR  x1,x1,#0xC0000000 ; sets the N and Z flags (bits 31,30)
MSR  NZCV, x1          ; copy x1 back into NZCV register to update the condition flags
```

#### Related concepts

[C1.1 Conditional instructions](#) on page C1-134

#### Related reference

[C1.4 Condition flags](#) on page C1-137

[C1.5 Updates to the condition flags in A32/T32 code](#) on page C1-138

[C1.10 Condition code suffixes and related flags](#) on page C1-143

[D1.8 Condition code suffixes and related flags](#) on page D1-655

## D1.4 Floating-point instructions that update the condition flags

All A64 floating-point comparison instructions can update the condition flags. These instructions update the flags directly in the NZCV register.

### **Related concepts**

[D1.5 Carry flag on page D1-652](#)

[D1.6 Overflow flag on page D1-653](#)

### **Related reference**

[D1.3 Updates to the condition flags in A64 code on page D1-650](#)

[C4.4 VCMP, VCMPE on page C4-605](#)

[C3.75 VMRS on page C3-524](#)

[C4.26 VMRS \(floating-point\) on page C4-627](#)

### **Related information**

[Arm Architecture Reference Manual](#)

## D1.5 Carry flag

The carry (C) flag is set when an operation results in a carry, or when a subtraction results in no borrow.

C is set in one of the following ways:

- For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction CMP and the negate instructions NEGS and NGCS, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For the integer and floating-point conditional compare instructions CCMP, CCMN, FCCMP, and FCCMPE, C and the other condition flags are set either to the result of the comparison, or directly from an immediate value.
- For the floating-point compare instructions, FCMP and FCMPE, C and the other condition flags are set to the result of the comparison.
- For other instructions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.

### *Related concepts*

[D1.6 Overflow flag on page D1-653](#)

### *Related reference*

[A2.7 Predeclared core register names in AArch32 state on page A2-61](#)

[A3.5 Predeclared core register names in AArch64 state on page A3-76](#)

[D1.8 Condition code suffixes and related flags on page D1-655](#)

[D1.3 Updates to the condition flags in A64 code on page D1-650](#)

## D1.6 Overflow flag

Overflow can occur for add, subtract, and compare operations.

In A64 instructions that use the 64-bit X registers, overflow occurs if the result of the operation is greater than or equal to  $2^{63}$ , or less than  $-2^{63}$ .

In A64 instructions that use the 32-bit W registers, overflow occurs if the result of the operation is greater than or equal to  $2^{31}$ , or less than  $-2^{31}$ .

### ***Related concepts***

[D1.5 Carry flag on page D1-652](#)

### ***Related reference***

[A2.7 Predeclared core register names in AArch32 state on page A2-61](#)

[D1.3 Updates to the condition flags in A64 code on page D1-650](#)

## D1.7 Condition code suffixes

Instructions that can be conditional have an optional two character condition code suffix.

Condition codes are shown in syntax descriptions as {cond}. The following table shows the condition codes that you can use:

Table D1-1 Condition code suffixes

Suffix	Meaning
EQ	Equal
NE	Not equal
CS	Carry set (identical to HS)
HS	Unsigned higher or same (identical to CS)
CC	Carry clear (identical to LO)
LO	Unsigned lower (identical to CC)
MI	Minus or negative result
PL	Positive or zero result
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (this is the default)

### Note

The meaning of some of these condition codes depends on whether the instruction that last updated the condition flags is a floating-point or integer instruction.

### Related reference

[C1.11 Comparison of condition code meanings in integer and floating-point code](#) on page C1-144

[C2.44 IT](#) on page C2-222

[C3.75 VMRS](#) on page C3-524

[C4.26 VMRS \(floating-point\)](#) on page C4-627

## D1.8 Condition code suffixes and related flags

Condition code suffixes define the conditions that must be met for the instruction to execute.

The following table shows the condition codes that you can use and the flag settings they depend on:

**Table D1-2 Condition code suffixes and related flags**

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned $\geq$ )
CC or LO	C clear	Lower (unsigned $<$ )
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned $>$ )
LS	C clear or Z set	Lower or same (unsigned $\leq$ )
GE	N and V the same	Signed $\geq$
LT	N and V differ	Signed $<$
GT	Z clear, N and V the same	Signed $>$
LE	Z set, N and V differ	Signed $\leq$
AL	Any	Always. This suffix is normally omitted.

The optional condition code is shown in syntax descriptions as `{cond}`. This condition is encoded in A32 instructions and in A64 instructions. For T32 instructions, the condition is encoded in a preceding IT instruction. An instruction with a condition code is only executed if the condition flags meet the specified condition.

The following is an example of conditional execution in A32 code:

```

ADD      r0, r1, r2      ; r0 = r1 + r2, don't update flags
ADDS     r0, r1, r2      ; r0 = r1 + r2, and update flags
ADDSCS   r0, r1, r2      ; If C flag set then r0 = r1 + r2,
                        ; and update flags
CMP      r0, r1          ; update flags based on r0-r1.

```

### Related reference

[C1.4 Condition flags on page C1-137](#)

[D1.3 Updates to the condition flags in A64 code on page D1-650](#)

[Chapter C2 A32 and T32 Instructions on page C2-151](#)

## D1.9 Optimization for execution speed

To optimize code for execution speed you must have detailed knowledge of the instruction timings, branch prediction logic, and cache behavior of your target system.

For more information, see the Technical Reference Manual for your processor.

### ***Related information***

*[Arm Architecture Reference Manual](#)*

*[Further reading](#)*

# Chapter D2

## A64 General Instructions

Describes the A64 general instructions.

It contains the following sections:

- [D2.1 A64 instructions in alphabetical order](#) on page D2-662.
- [D2.2 Register restrictions for A64 instructions](#) on page D2-669.
- [D2.3 ADC](#) on page D2-670.
- [D2.4 ADCS](#) on page D2-671.
- [D2.5 ADD \(extended register\)](#) on page D2-672.
- [D2.6 ADD \(immediate\)](#) on page D2-674.
- [D2.7 ADD \(shifted register\)](#) on page D2-675.
- [D2.8 ADDG](#) on page D2-676.
- [D2.9 ADDS \(extended register\)](#) on page D2-677.
- [D2.10 ADDS \(immediate\)](#) on page D2-679.
- [D2.11 ADDS \(shifted register\)](#) on page D2-680.
- [D2.12 ADR](#) on page D2-681.
- [D2.13 ADRP](#) on page D2-682.
- [D2.14 AND \(immediate\)](#) on page D2-683.
- [D2.15 AND \(shifted register\)](#) on page D2-684.
- [D2.16 ANDS \(immediate\)](#) on page D2-685.
- [D2.17 ANDS \(shifted register\)](#) on page D2-686.
- [D2.18 ASR \(register\)](#) on page D2-687.
- [D2.19 ASR \(immediate\)](#) on page D2-688.
- [D2.20 ASRV](#) on page D2-689.
- [D2.21 AT](#) on page D2-690.
- [D2.22 AUTDA, AUTDZA](#) on page D2-692.
- [D2.23 AUTDB, AUTDZB](#) on page D2-693.

- *D2.24 AUTIA, AUTIZA, AUTIA1716, AUTIASP, AUTIAZ* on page D2-694.
- *D2.25 AUTIB, AUTIZB, AUTIB1716, AUTIBSP, AUTIBZ* on page D2-695.
- *D2.26 AXFlag* on page D2-696.
- *D2.27 B.cond* on page D2-697.
- *D2.28 B* on page D2-698.
- *D2.29 BFC* on page D2-699.
- *D2.30 BFI* on page D2-700.
- *D2.31 BFM* on page D2-701.
- *D2.32 BFXIL* on page D2-702.
- *D2.33 BIC (shifted register)* on page D2-703.
- *D2.34 BICS (shifted register)* on page D2-704.
- *D2.35 BL* on page D2-705.
- *D2.36 BLR* on page D2-706.
- *D2.37 BLRAA, BLRAAZ, BLRAB, BLRABZ* on page D2-707.
- *D2.38 BR* on page D2-708.
- *D2.39 BRAA, BRAAZ, BRAB, BRABZ* on page D2-709.
- *D2.40 BRK* on page D2-710.
- *D2.41 BTI* on page D2-711.
- *D2.42 CBNZ* on page D2-712.
- *D2.43 CBZ* on page D2-713.
- *D2.44 CCMN (immediate)* on page D2-714.
- *D2.45 CCMN (register)* on page D2-715.
- *D2.46 CCMP (immediate)* on page D2-716.
- *D2.47 CCMP (register)* on page D2-717.
- *D2.48 CINC* on page D2-718.
- *D2.49 CINV* on page D2-719.
- *D2.50 CLREX* on page D2-720.
- *D2.51 CLS* on page D2-721.
- *D2.52 CLZ* on page D2-722.
- *D2.53 CMN (extended register)* on page D2-723.
- *D2.54 CMN (immediate)* on page D2-725.
- *D2.55 CMN (shifted register)* on page D2-726.
- *D2.56 CMP (extended register)* on page D2-727.
- *D2.57 CMP (immediate)* on page D2-729.
- *D2.58 CMP (shifted register)* on page D2-730.
- *D2.59 CMPP* on page D2-731.
- *D2.60 CNEG* on page D2-732.
- *D2.61 CRC32B, CRC32H, CRC32W, CRC32X* on page D2-733.
- *D2.62 CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page D2-734.
- *D2.63 CSDB* on page D2-735.
- *D2.64 CSEL* on page D2-737.
- *D2.65 CSET* on page D2-738.
- *D2.66 CSETM* on page D2-739.
- *D2.67 CSINC* on page D2-740.
- *D2.68 CSINV* on page D2-741.
- *D2.69 CSNEG* on page D2-742.
- *D2.70 DC* on page D2-743.
- *D2.71 DCPS1* on page D2-744.
- *D2.72 DCPS2* on page D2-745.
- *D2.73 DCPS3* on page D2-746.
- *D2.74 DMB* on page D2-747.
- *D2.75 DRPS* on page D2-749.
- *D2.76 DSB* on page D2-750.
- *D2.77 EON (shifted register)* on page D2-752.
- *D2.78 EOR (immediate)* on page D2-753.
- *D2.79 EOR (shifted register)* on page D2-754.

- *D2.80 ERET* on page D2-755.
- *D2.81 ERETA<sub>A</sub>, ERETAB* on page D2-756.
- *D2.82 ESB* on page D2-757.
- *D2.83 EXTR* on page D2-758.
- *D2.84 GMI* on page D2-759.
- *D2.85 HINT* on page D2-760.
- *D2.86 HLT* on page D2-761.
- *D2.87 HVC* on page D2-762.
- *D2.88 IC* on page D2-763.
- *D2.89 IRG* on page D2-764.
- *D2.90 ISB* on page D2-765.
- *D2.91 LDG* on page D2-766.
- *D2.92 LDGV* on page D2-767.
- *D2.93 LSL (register)* on page D2-768.
- *D2.94 LSL (immediate)* on page D2-769.
- *D2.95 LSLV* on page D2-770.
- *D2.96 LSR (register)* on page D2-771.
- *D2.97 LSR (immediate)* on page D2-772.
- *D2.98 LSRV* on page D2-773.
- *D2.99 MADD* on page D2-774.
- *D2.100 MNEG* on page D2-775.
- *D2.101 MOV (to or from SP)* on page D2-776.
- *D2.102 MOV (inverted wide immediate)* on page D2-777.
- *D2.103 MOV (wide immediate)* on page D2-778.
- *D2.104 MOV (bitmask immediate)* on page D2-779.
- *D2.105 MOV (register)* on page D2-780.
- *D2.106 MOVK* on page D2-781.
- *D2.107 MOVN* on page D2-782.
- *D2.108 MOVZ* on page D2-783.
- *D2.109 MRS* on page D2-784.
- *D2.110 MSR (immediate)* on page D2-785.
- *D2.111 MSR (register)* on page D2-786.
- *D2.112 MSUB* on page D2-787.
- *D2.113 MUL* on page D2-788.
- *D2.114 MVN* on page D2-789.
- *D2.115 NEG (shifted register)* on page D2-790.
- *D2.116 NEGS* on page D2-791.
- *D2.117 NGC* on page D2-792.
- *D2.118 NGCS* on page D2-793.
- *D2.119 NOP* on page D2-794.
- *D2.120 ORN (shifted register)* on page D2-795.
- *D2.121 ORR (immediate)* on page D2-796.
- *D2.122 ORR (shifted register)* on page D2-797.
- *D2.123 PACDA, PACDZA* on page D2-798.
- *D2.124 PACDB, PACDZB* on page D2-799.
- *D2.125 PACGA* on page D2-800.
- *D2.126 PACIA, PACIZA, PACIA<sub>1716</sub>, PACIASP, PACIAZ* on page D2-801.
- *D2.127 PACIB, PACIZB, PACIB<sub>1716</sub>, PACIBSP, PACIBZ* on page D2-802.
- *D2.128 PSB* on page D2-803.
- *D2.129 RBIT* on page D2-804.
- *D2.130 RET* on page D2-805.
- *D2.131 RETAA, RETAB* on page D2-806.
- *D2.132 REV<sub>16</sub>* on page D2-807.
- *D2.133 REV32* on page D2-808.
- *D2.134 REV64* on page D2-809.
- *D2.135 REV* on page D2-810.

- *D2.136 ROR (immediate)* on page D2-811.
- *D2.137 ROR (register)* on page D2-812.
- *D2.138 RORV* on page D2-813.
- *D2.139 SBC* on page D2-814.
- *D2.140 SBCS* on page D2-815.
- *D2.141 SBFIZ* on page D2-816.
- *D2.142 SBFM* on page D2-817.
- *D2.143 SBFX* on page D2-818.
- *D2.144 SDIV* on page D2-819.
- *D2.145 SEV* on page D2-820.
- *D2.146 SEVL* on page D2-821.
- *D2.147 SMADDL* on page D2-822.
- *D2.148 SMC* on page D2-823.
- *D2.149 SMNEGL* on page D2-824.
- *D2.150 SMSUBL* on page D2-825.
- *D2.151 SMULH* on page D2-826.
- *D2.152 SMULL* on page D2-827.
- *D2.153 ST2G* on page D2-828.
- *D2.154 STG* on page D2-829.
- *D2.155 STGP* on page D2-830.
- *D2.156 STGV* on page D2-831.
- *D2.157 STZ2G* on page D2-832.
- *D2.158 STZG* on page D2-833.
- *D2.159 SUB (extended register)* on page D2-834.
- *D2.160 SUB (immediate)* on page D2-836.
- *D2.161 SUB (shifted register)* on page D2-837.
- *D2.162 SUBG* on page D2-838.
- *D2.163 SUBP* on page D2-839.
- *D2.164 SUBPS* on page D2-840.
- *D2.165 SUBS (extended register)* on page D2-841.
- *D2.166 SUBS (immediate)* on page D2-843.
- *D2.167 SUBS (shifted register)* on page D2-844.
- *D2.168 SVC* on page D2-845.
- *D2.169 SXTB* on page D2-846.
- *D2.170 SXTH* on page D2-847.
- *D2.171 SXTW* on page D2-848.
- *D2.172 SYS* on page D2-849.
- *D2.173 SYSL* on page D2-850.
- *D2.174 TBNZ* on page D2-851.
- *D2.175 TBZ* on page D2-852.
- *D2.176 TLBI* on page D2-853.
- *D2.177 TST (immediate)* on page D2-855.
- *D2.178 TST (shifted register)* on page D2-856.
- *D2.179 UBFIZ* on page D2-857.
- *D2.180 UBFM* on page D2-858.
- *D2.181 UBFX* on page D2-859.
- *D2.182 UDIV* on page D2-860.
- *D2.183 UMADDL* on page D2-861.
- *D2.184 UMNEGL* on page D2-862.
- *D2.185 UMSUBL* on page D2-863.
- *D2.186 UMULH* on page D2-864.
- *D2.187 UMULL* on page D2-865.
- *D2.188 UXTB* on page D2-866.
- *D2.189 UXTH* on page D2-867.
- *D2.190 XAFlag* on page D2-868.
- *D2.191 WFE* on page D2-869.

- *D2.192 WFI* on page D2-870.
- *D2.193 XPACD, XPACI, XPACLRI* on page D2-871.
- *D2.194 YIELD* on page D2-872.

## D2.1 A64 instructions in alphabetical order

A summary of the A64 instructions and pseudo-instructions that are supported.

**Table D2-1 Summary of A64 general instructions**

Mnemonic	Brief description	See
ADC	Add with Carry	<a href="#">D2.3 ADC on page D2-670</a>
ADCS	Add with Carry, setting flags	<a href="#">D2.4 ADCS on page D2-671</a>
ADD (extended register)	Add (extended register)	<a href="#">D2.5 ADD (extended register) on page D2-672</a>
ADD (immediate)	Add (immediate)	<a href="#">D2.6 ADD (immediate) on page D2-674</a>
ADD (shifted register)	Add (shifted register)	<a href="#">D2.7 ADD (shifted register) on page D2-675</a>
ADDG	Add with Tag	<a href="#">D2.8 ADDG on page D2-676</a>
ADDS (extended register)	Add (extended register), setting flags	<a href="#">D2.9 ADDS (extended register) on page D2-677</a>
ADDS (immediate)	Add (immediate), setting flags	<a href="#">D2.10 ADDS (immediate) on page D2-679</a>
ADDS (shifted register)	Add (shifted register), setting flags	<a href="#">D2.11 ADDS (shifted register) on page D2-680</a>
ADR	Form PC-relative address	<a href="#">D2.12 ADR on page D2-681</a>
ADRL pseudo-instruction	Load a PC-relative address into a register	
ADRP	Form PC-relative address to 4KB page	<a href="#">D2.13 ADRP on page D2-682</a>
AND (immediate)	Bitwise AND (immediate)	<a href="#">D2.14 AND (immediate) on page D2-683</a>
AND (shifted register)	Bitwise AND (shifted register)	<a href="#">D2.15 AND (shifted register) on page D2-684</a>
ANDS (immediate)	Bitwise AND (immediate), setting flags	<a href="#">D2.16 ANDS (immediate) on page D2-685</a>
ANDS (shifted register)	Bitwise AND (shifted register), setting flags	<a href="#">D2.17 ANDS (shifted register) on page D2-686</a>
ASR (register)	Arithmetic Shift Right (register)	<a href="#">D2.18 ASR (register) on page D2-687</a>
ASR (immediate)	Arithmetic Shift Right (immediate)	<a href="#">D2.19 ASR (immediate) on page D2-688</a>
ASRV	Arithmetic Shift Right Variable	<a href="#">D2.20 ASRV on page D2-689</a>
AT	Address Translate	<a href="#">D2.21 AT on page D2-690</a>
AUTDA, AUTDZA	Authenticate Data address, using key A	<a href="#">D2.22 AUTDA, AUTDZA on page D2-692</a>
AUTDB, AUTDZB	Authenticate Data address, using key B	<a href="#">D2.23 AUTDB, AUTDZB on page D2-693</a>
AUTIA, AUTIZA, AUTIA1716, AUTIASP, AUTIAZ	Authenticate Instruction address, using key A	<a href="#">D2.24 AUTIA, AUTIZA, AUTIA1716, AUTIASP, AUTIAZ on page D2-694</a>
AUTIB, AUTIZB, AUTIB1716, AUTIBSP, AUTIBZ	Authenticate Instruction address, using key B	<a href="#">D2.25 AUTIB, AUTIZB, AUTIB1716, AUTIBSP, AUTIBZ on page D2-695</a>
AXFlag	Convert floating-point condition flags from Arm to external format	<a href="#">D2.26 AXFlag on page D2-696</a>
B.cond	Branch conditionally	<a href="#">D2.27 B.cond on page D2-697</a>
B	Branch	<a href="#">D2.28 B on page D2-698</a>

**Table D2-1 Summary of A64 general instructions (continued)**

Mnemonic	Brief description	See
BFC	Bitfield Clear, leaving other bits unchanged	<a href="#">D2.29 BFC on page D2-699</a>
BFI	Bitfield Insert	<a href="#">D2.30 BFI on page D2-700</a>
BFM	Bitfield Move	<a href="#">D2.31 BFM on page D2-701</a>
BFXIL	Bitfield extract and insert at low end	<a href="#">D2.32 BFXIL on page D2-702</a>
BIC (shifted register)	Bitwise Bit Clear (shifted register)	<a href="#">D2.33 BIC (shifted register) on page D2-703</a>
BICS (shifted register)	Bitwise Bit Clear (shifted register), setting flags	<a href="#">D2.34 BICS (shifted register) on page D2-704</a>
BL	Branch with Link	<a href="#">D2.35 BL on page D2-705</a>
BLR	Branch with Link to Register	<a href="#">D2.36 BLR on page D2-706</a>
BLRAA, BLRAAZ, BLRAB, BLRABZ	Branch with Link to Register, with pointer authentication	<a href="#">D2.37 BLRAA, BLRAAZ, BLRAB, BLRABZ on page D2-707</a>
BR	Branch to Register	<a href="#">D2.38 BR on page D2-708</a>
BRAA, BRAAZ, BRAB, BRABZ	Branch to Register, with pointer authentication	<a href="#">D2.39 BRAA, BRAAZ, BRAB, BRABZ on page D2-709</a>
BRK	Breakpoint instruction	<a href="#">D2.40 BRK on page D2-710</a>
BTI	Branch Target Identification	<a href="#">D2.41 BTI on page D2-711</a>
CBNZ	Compare and Branch on Nonzero	<a href="#">D2.42 CBNZ on page D2-712</a>
CBZ	Compare and Branch on Zero	<a href="#">D2.43 CBZ on page D2-713</a>
CCMN (immediate)	Conditional Compare Negative (immediate)	<a href="#">D2.44 CCMN (immediate) on page D2-714</a>
CCMN (register)	Conditional Compare Negative (register)	<a href="#">D2.45 CCMN (register) on page D2-715</a>
CCMP (immediate)	Conditional Compare (immediate)	<a href="#">D2.46 CCMP (immediate) on page D2-716</a>
CCMP (register)	Conditional Compare (register)	<a href="#">D2.47 CCMP (register) on page D2-717</a>
CINC	Conditional Increment	<a href="#">D2.48 CINC on page D2-718</a>
CINV	Conditional Invert	<a href="#">D2.49 CINV on page D2-719</a>
CLREX	Clear Exclusive	<a href="#">D2.50 CLREX on page D2-720</a>
CLS	Count leading sign bits	<a href="#">D2.51 CLS on page D2-721</a>
CLZ	Count leading zero bits	<a href="#">D2.52 CLZ on page D2-722</a>
CMN (extended register)	Compare Negative (extended register)	<a href="#">D2.53 CMN (extended register) on page D2-723</a>
CMN (immediate)	Compare Negative (immediate)	<a href="#">D2.54 CMN (immediate) on page D2-725</a>
CMN (shifted register)	Compare Negative (shifted register)	<a href="#">D2.55 CMN (shifted register) on page D2-726</a>
CMP (extended register)	Compare (extended register)	<a href="#">D2.56 CMP (extended register) on page D2-727</a>
CMP (immediate)	Compare (immediate)	<a href="#">D2.57 CMP (immediate) on page D2-729</a>
CMP (shifted register)	Compare (shifted register)	<a href="#">D2.58 CMP (shifted register) on page D2-730</a>
CMPP	Compare with Tag	<a href="#">D2.59 CMPP on page D2-731</a>
CNEG	Conditional Negate	<a href="#">D2.60 CNEG on page D2-732</a>

**Table D2-1 Summary of A64 general instructions (continued)**

Mnemonic	Brief description	See
CRC32B, CRC32H, CRC32W, CRC32X	CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register	<a href="#">D2.61 CRC32B, CRC32H, CRC32W, CRC32X on page D2-733</a>
CRC32CB, CRC32CH, CRC32CW, CRC32CX	CRC32C checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register	<a href="#">D2.62 CRC32CB, CRC32CH, CRC32CW, CRC32CX on page D2-734</a>
CSDB	Consumption of Speculative Data Barrier	<a href="#">D2.63 CSDB on page D2-735</a>
CSEL	Conditional Select	<a href="#">D2.64 CSEL on page D2-737</a>
CSET	Conditional Set	<a href="#">D2.65 CSET on page D2-738</a>
CSETM	Conditional Set Mask	<a href="#">D2.66 CSETM on page D2-739</a>
CSINC	Conditional Select Increment	<a href="#">D2.67 CSINC on page D2-740</a>
CSINV	Conditional Select Invert	<a href="#">D2.68 CSINV on page D2-741</a>
CSNEG	Conditional Select Negation	<a href="#">D2.69 CSNEG on page D2-742</a>
DC	Data Cache operation	<a href="#">D2.70 DC on page D2-743</a>
DCPS1	Debug Change PE State to EL1	<a href="#">D2.71 DCPS1 on page D2-744</a>
DCPS2	Debug Change PE State to EL2	<a href="#">D2.72 DCPS2 on page D2-745</a>
DCPS3	Debug Change PE State to EL3	<a href="#">D2.73 DCPS3 on page D2-746</a>
DMB	Data Memory Barrier	<a href="#">D2.74 DMB on page D2-747</a>
DRPS	Debug restore process state	<a href="#">D2.75 DRPS on page D2-749</a>
DSB	Data Synchronization Barrier	<a href="#">D2.76 DSB on page D2-750</a>
EON (shifted register)	Bitwise Exclusive OR NOT (shifted register)	<a href="#">D2.77 EON (shifted register) on page D2-752</a>
EOR (immediate)	Bitwise Exclusive OR (immediate)	<a href="#">D2.78 EOR (immediate) on page D2-753</a>
EOR (shifted register)	Bitwise Exclusive OR (shifted register)	<a href="#">D2.79 EOR (shifted register) on page D2-754</a>
ERET	Returns from an exception	<a href="#">D2.80 ERET on page D2-755</a>
ERETAA, ERETAB	Exception Return, with pointer authentication	<a href="#">D2.81 ERETAAC, ERETAB on page D2-756</a>
ESB	Error Synchronization Barrier	<a href="#">D2.82 ESB on page D2-757</a>
EXTR	Extract register	<a href="#">D2.83 EXTR on page D2-758</a>
HINT	Hint instruction	<a href="#">D2.85 HINT on page D2-760</a>
HLT	Halt instruction	<a href="#">D2.86 HLT on page D2-761</a>
HVC	Hypervisor call to allow OS code to call the Hypervisor	<a href="#">D2.87 HVC on page D2-762</a>
IC	Instruction Cache operation	<a href="#">D2.88 IC on page D2-763</a>
IRG	Insert Random Tag	<a href="#">D2.89 IRG on page D2-764</a>
ISB	Instruction Synchronization Barrier	<a href="#">D2.90 ISB on page D2-765</a>
LDG	Load Allocation Tag	<a href="#">D2.91 LDG on page D2-766</a>

**Table D2-1 Summary of A64 general instructions (continued)**

Mnemonic	Brief description	See
LDGV	Load Allocation Tag	<a href="#">D2.92 LDGV on page D2-767</a>
LSL (register)	Logical Shift Left (register)	<a href="#">D2.93 LSL (register) on page D2-768</a>
LSL (immediate)	Logical Shift Left (immediate)	<a href="#">D2.94 LSL (immediate) on page D2-769</a>
LSLV	Logical Shift Left Variable	<a href="#">D2.95 LSLV on page D2-770</a>
LSR (register)	Logical Shift Right (register)	<a href="#">D2.96 LSR (register) on page D2-771</a>
LSR (immediate)	Logical Shift Right (immediate)	<a href="#">D2.97 LSR (immediate) on page D2-772</a>
LSRV	Logical Shift Right Variable	<a href="#">D2.98 LSrv on page D2-773</a>
MADD	Multiply-Add	<a href="#">D2.99 MADD on page D2-774</a>
MNEG	Multiply-Negate	<a href="#">D2.100 MNEG on page D2-775</a>
MOV (to or from SP)	Move between register and stack pointer	<a href="#">D2.101 MOV (to or from SP) on page D2-776</a>
MOV (inverted wide immediate)	Move (inverted wide immediate)	<a href="#">D2.102 MOV (inverted wide immediate) on page D2-777</a>
MOV (wide immediate)	Move (wide immediate)	<a href="#">D2.103 MOV (wide immediate) on page D2-778</a>
MOV (bitmask immediate)	Move (bitmask immediate)	<a href="#">D2.104 MOV (bitmask immediate) on page D2-779</a>
MOV (register)	Move (register)	<a href="#">D2.105 MOV (register) on page D2-780</a>
MOVK	Move wide with keep	<a href="#">D2.106 MOVK on page D2-781</a>
MOVL pseudo-instruction	Load a register with either a 32-bit or 64-bit immediate value or any address	
MOVN	Move wide with NOT	<a href="#">D2.107 MOVN on page D2-782</a>
MOVZ	Move wide with zero	<a href="#">D2.108 MOVZ on page D2-783</a>
MRS	Move System Register	<a href="#">D2.109 MRS on page D2-784</a>
MSR (immediate)	Move immediate value to Special Register	<a href="#">D2.110 MSR (immediate) on page D2-785</a>
MSR (register)	Move general-purpose register to System Register	<a href="#">D2.111 MSR (register) on page D2-786</a>
MSUB	Multiply-Subtract	<a href="#">D2.112 MSUB on page D2-787</a>
MUL	Multiply	<a href="#">D2.113 MUL on page D2-788</a>
MVN	Bitwise NOT	<a href="#">D2.114 MVN on page D2-789</a>
NEG (shifted register)	Negate (shifted register)	<a href="#">D2.115 NEG (shifted register) on page D2-790</a>
NEGS	Negate, setting flags	<a href="#">D2.116 NEGS on page D2-791</a>
NGC	Negate with Carry	<a href="#">D2.117 NGC on page D2-792</a>
NGCS	Negate with Carry, setting flags	<a href="#">D2.118 NGCS on page D2-793</a>
NOP	No Operation	<a href="#">D2.119 NOP on page D2-794</a>
ORN (shifted register)	Bitwise OR NOT (shifted register)	<a href="#">D2.120 ORN (shifted register) on page D2-795</a>
ORR (immediate)	Bitwise OR (immediate)	<a href="#">D2.121 ORR (immediate) on page D2-796</a>
ORR (shifted register)	Bitwise OR (shifted register)	<a href="#">D2.122 ORR (shifted register) on page D2-797</a>

**Table D2-1 Summary of A64 general instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
PACDA, PACDZA	Pointer Authentication Code for Data address, using key A	<a href="#">D2.123 PACDA, PACDZA on page D2-798</a>
PACDB, PACDZB	Pointer Authentication Code for Data address, using key B	<a href="#">D2.124 PACDB, PACDZB on page D2-799</a>
PACGA	Pointer Authentication Code, using Generic key	<a href="#">D2.125 PACG4 on page D2-800</a>
PACIA, PACIZA, PACIA1716, PACIASP, PACIAZ	Pointer Authentication Code for Instruction address, using key A	<a href="#">D2.126 PACIA, PACIZA, PACIA1716, PACIASP, PACIAZ on page D2-801</a>
PACIB, PACIZB, PACIB1716, PACIBSP, PACIBZ	Pointer Authentication Code for Instruction address, using key B	<a href="#">D2.127 PACIB, PACIZB, PACIB1716, PACIBSP, PACIBZ on page D2-802</a>
PSB	Profiling Synchronization Barrier	<a href="#">D2.128 PSB on page D2-803</a>
RBIT	Reverse Bits	<a href="#">D2.129 RBIT on page D2-804</a>
RET	Return from subroutine	<a href="#">D2.130 RET on page D2-805</a>
RETA4, RETAB	Return from subroutine, with pointer authentication	<a href="#">D2.131 RETAA, RETAB on page D2-806</a>
REV16	Reverse bytes in 16-bit halfwords	<a href="#">D2.132 REV16 on page D2-807</a>
REV32	Reverse bytes in 32-bit words	<a href="#">D2.133 REV32 on page D2-808</a>
REV64	Reverse Bytes	<a href="#">D2.134 REV64 on page D2-809</a>
REV	Reverse Bytes	<a href="#">D2.135 REV on page D2-810</a>
ROR (immediate)	Rotate right (immediate)	<a href="#">D2.136 ROR (immediate) on page D2-811</a>
ROR (register)	Rotate Right (register)	<a href="#">D2.137 ROR (register) on page D2-812</a>
RORV	Rotate Right Variable	<a href="#">D2.138 RORV on page D2-813</a>
SBC	Subtract with Carry	<a href="#">D2.139 SBC on page D2-814</a>
SBCS	Subtract with Carry, setting flags	<a href="#">D2.140 SBCS on page D2-815</a>
SBFIZ	Signed Bitfield Insert in Zero	<a href="#">D2.141 SBFIZ on page D2-816</a>
SBFM	Signed Bitfield Move	<a href="#">D2.142 SBFM on page D2-817</a>
SBFX	Signed Bitfield Extract	<a href="#">D2.143 SBFX on page D2-818</a>
SDIV	Signed Divide	<a href="#">D2.144 SDIV on page D2-819</a>
SEV	Send Event	<a href="#">D2.145 SEV on page D2-820</a>
SEVL	Send Event Local	<a href="#">D2.146 SEVL on page D2-821</a>
SMADDL	Signed Multiply-Add Long	<a href="#">D2.147 SMADDL on page D2-822</a>
SMC	Supervisor call to allow OS or Hypervisor code to call the Secure Monitor	<a href="#">D2.148 SMC on page D2-823</a>
SMNEGL	Signed Multiply-Negate Long	<a href="#">D2.149 SMNEGL on page D2-824</a>
SMSUBL	Signed Multiply-Subtract Long	<a href="#">D2.150 SMSUBL on page D2-825</a>
SMULH	Signed Multiply High	<a href="#">D2.151 SMULH on page D2-826</a>

**Table D2-1 Summary of A64 general instructions (continued)**

Mnemonic	Brief description	See
SMULL	Signed Multiply Long	<a href="#">D2.152 SMULL on page D2-827</a>
ST2G	Store Allocation Tags	<a href="#">D2.153 ST2G on page D2-828</a>
STG	Store Allocation Tag	<a href="#">D2.154 STG on page D2-829</a>
STGP	Store Allocation Tag and Pair of registers.	<a href="#">D2.155 STGP on page D2-830</a>
STGV	Store Tag Vector	<a href="#">D2.156 STGV on page D2-831</a>
STZ2G	Store Allocation Tags, Zeroing	<a href="#">D2.157 STZ2G on page D2-832</a>
STZG	Store Allocation Tag, Zeroing	<a href="#">D2.158 STZG on page D2-833</a>
SUB (extended register)	Subtract (extended register)	<a href="#">D2.159 SUB (extended register) on page D2-834</a>
SUB (immediate)	Subtract (immediate)	<a href="#">D2.160 SUB (immediate) on page D2-836</a>
SUB (shifted register)	Subtract (shifted register)	<a href="#">D2.161 SUB (shifted register) on page D2-837</a>
SUBG	Subtract with Tag	<a href="#">D2.162 SUBG on page D2-838</a>
SUBP	Subtract Pointer	<a href="#">D2.163 SUBP on page D2-839</a>
SUBPS	Subtract Pointer, setting Flags	<a href="#">D2.164 SUBPS on page D2-840</a>
SUBS (extended register)	Subtract (extended register), setting flags	<a href="#">D2.165 SUBS (extended register) on page D2-841</a>
SUBS (immediate)	Subtract (immediate), setting flags	<a href="#">D2.166 SUBS (immediate) on page D2-843</a>
SUBS (shifted register)	Subtract (shifted register), setting flags	<a href="#">D2.167 SUBS (shifted register) on page D2-844</a>
SVC	Supervisor call to allow application code to call the OS	<a href="#">D2.168 SVC on page D2-845</a>
SXTB	Signed Extend Byte	<a href="#">D2.169 SXTB on page D2-846</a>
SXTH	Sign Extend Halfword	<a href="#">D2.170 SXTH on page D2-847</a>
SXTW	Sign Extend Word	<a href="#">D2.171 SXTW on page D2-848</a>
SYS	System instruction	<a href="#">D2.172 SYS on page D2-849</a>
SYSL	System instruction with result	<a href="#">D2.173 SYSL on page D2-850</a>
TBNZ	Test bit and Branch if Nonzero	<a href="#">D2.174 TBNZ on page D2-851</a>
TBZ	Test bit and Branch if Zero	<a href="#">D2.175 TBZ on page D2-852</a>
TLBI	TLB Invalidate operation	<a href="#">D2.176 TLBI on page D2-853</a>
TST (immediate)	, setting the condition flags and discarding the result	<a href="#">D2.177 TST (immediate) on page D2-855</a>
TST (shifted register)	Test (shifted register)	<a href="#">D2.178 TST (shifted register) on page D2-856</a>
UBFIZ	Unsigned Bitfield Insert in Zero	<a href="#">D2.179 UBFIZ on page D2-857</a>
UBFM	Unsigned Bitfield Move	<a href="#">D2.180 UBFM on page D2-858</a>
UBFX	Unsigned Bitfield Extract	<a href="#">D2.181 UBFX on page D2-859</a>
UDIV	Unsigned Divide	<a href="#">D2.182 UDIV on page D2-860</a>
UMADDL	Unsigned Multiply-Add Long	<a href="#">D2.183 UMADDL on page D2-861</a>
UMNEGL	Unsigned Multiply-Negate Long	<a href="#">D2.184 UMNEGL on page D2-862</a>

**Table D2-1 Summary of A64 general instructions (continued)**

Mnemonic	Brief description	See
UMSUBL	Unsigned Multiply-Subtract Long	<a href="#">D2.185 UMSUBL on page D2-863</a>
UMULH	Unsigned Multiply High	<a href="#">D2.186 UMULH on page D2-864</a>
UMULL	Unsigned Multiply Long	<a href="#">D2.187 UMULL on page D2-865</a>
UXTB	Unsigned Extend Byte	<a href="#">D2.188 UXTB on page D2-866</a>
UXTH	Unsigned Extend Halfword	<a href="#">D2.189 UXTH on page D2-867</a>
WFE	Wait For Event	<a href="#">D2.191 WFE on page D2-869</a>
WFI	Wait For Interrupt	<a href="#">D2.192 WFI on page D2-870</a>
XAFlag	Convert floating-point condition flags from external format to Arm format	<a href="#">D2.190 XAFlag on page D2-868</a>
XPACD, XPACI, XPAACLRI	Strip Pointer Authentication Code	<a href="#">D2.193 XPACD, XPACI, XPAACLRI on page D2-871</a>
YIELD	YIELD	<a href="#">D2.194 YIELD on page D2-872</a>

## D2.2 Register restrictions for A64 instructions

In A64 instructions, the general-purpose integer registers are W0-W30 for 32-bit registers and X0-X30 for 64-bit registers.

You cannot refer to register 31 by number. In a few instructions, you can refer to it using one of the following names:

**WSP**

the current stack pointer in a 32-bit context.

**SP**

the current stack pointer in a 64-bit context.

**WZR**

the zero register in a 32-bit context.

**XZR**

the zero register in a 64-bit context.

You can only use one of these names if it is mentioned in the Syntax section for the instruction.

You cannot refer to the Program Counter (PC) explicitly by name or by number.

## D2.3 ADC

Add with Carry.

### Syntax

ADC *Wd*, *Wn*, *Wm* ; 32-bit

ADC *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

### Operation

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

$Rd = Rn + Rm + C$ , where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.4 ADCS

Add with Carry, setting flags.

### Syntax

ADCS *Wd*, *Wn*, *Wm* ; 32-bit

ADCS *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

### Operation

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.

$Rd = Rn + Rm + C$ , where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.5 ADD (extended register)

Add (extended register).

### Syntax

ADD *Wd/WSP*, *Wn/WSP*, *Wm*{, *extend* {#*amount*}} ; 32-bit

ADD *Xd/SP*, *Xn/SP*, *Rm*{, *extend* {#*amount*}} ; 64-bit

Where:

***Wd/WSP***

Is the 32-bit name of the destination general-purpose register or stack pointer.

***Wn/WSP***

Is the 32-bit name of the first source general-purpose register or stack pointer.

***Wm***

Is the 32-bit name of the second general-purpose source register.

***extend***

Is the extension to be applied to the second source operand:

#### 32-bit general registers

Can be one of UXTB, UXTH, LSL|UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rd* or *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

#### 64-bit general registers

Can be one of UXTB, UXTH, UXTW, LSL|UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rd* or *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

***Xd/SP***

Is the 64-bit name of the destination general-purpose register or stack pointer.

***Xn/SP***

Is the 64-bit name of the first source general-purpose register or stack pointer.

***R***

Is a width specifier, and can be either *W* or *X*.

***m***

Is the number [0-30] of the second general-purpose source register or the name ZR (31).

***amount***

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. 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

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.

$Rd = Rn + LSL(extend(Rm), amount)$ , where *R* is either *W* or *X*.

## Usage

**Table D2-2 ADD (64-bit general registers) specifier combinations**

<i>R</i>	<i>extend</i>
W	SXTB
W	SXTH
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL   UXTX
X	SXTX

### *Related reference*

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.6 ADD (immediate)

Add (immediate).

This instruction is used by the alias MOV (to or from SP).

### Syntax

ADD *Wd/WSP*, *Wn/WSP*, #*imm{, shift}* ; 32-bit

ADD *Xd/SP*, *Xn/SP*, #*imm{, shift}* ; 64-bit

Where:

***Wd/WSP***

Is the 32-bit name of the destination general-purpose register or stack pointer.

***Wn/WSP***

Is the 32-bit name of the source general-purpose register or stack pointer.

***Xd/SP***

Is the 64-bit name of the destination general-purpose register or stack pointer.

***Xn/SP***

Is the 64-bit name of the source general-purpose register or stack pointer.

***imm***

Is an unsigned immediate, in the range 0 to 4095.

***shift***

Is the optional left shift to apply to the immediate, defaulting to LSL #0, and can be either LSL #0 or LSL #12.

### Operation

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

$Rd = Rn + shift(imm)$ , where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.7 ADD (shifted register)

Add (shifted register).

### Syntax

ADD *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

ADD *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

### Operation

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

$Rd = Rn + \text{shift}(Rm, amount)$ , where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.8 ADDG

Add with Tag.

### Syntax

ADDG *Xd/SP*, *Xn/SP*, #<uimm6>, #<uimm4>

Where:

***Xd/SP***

Is the 64-bit name of the destination general-purpose register or stack pointer.

***Xn/SP***

Is the 64-bit name of the source general-purpose register or stack pointer.

**<uimm6>**

Is an unsigned immediate, a multiple of 16 in the range 0 to 1008.

**<uimm4>**

Is an unsigned immediate, in the range 0 to 15.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR\_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.9 ADDS (extended register)

Add (extended register), setting flags.

This instruction is used by the alias CMN (extended register).

### Syntax

`ADDS Wd, Wn/WSP, Wm{, extend {#amount}} ; 32-bit`

`ADDS Xd, Xn/SP, Rm{, extend {#amount}} ; 64-bit`

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Wn/WSP**

Is the 32-bit name of the first source general-purpose register or stack pointer.

**Wm**

Is the 32-bit name of the second general-purpose source register.

**extend**

Is the extension to be applied to the second source operand:

#### 32-bit general registers

Can be one of UXTB, UXTH, LSL | UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

#### 64-bit general registers

Can be one of UXTB, UXTH, UXTW, LSL | UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Xn/SP**

Is the 64-bit name of the first source general-purpose register or stack pointer.

**R**

Is a width specifier, and can be either W or X.

**m**

Is the number [0-30] of the second general-purpose source register or the name ZR (31).

**amount**

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. 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

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.

$Rd = Rn + LSL(\text{extend}(Rm), \text{amount})$ , where *R* is either W or X.

## Usage

Table D2-3 ADDS (64-bit general registers) specifier combinations

<i>R</i>	<i>extend</i>
W	SXTB
W	SXTH
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL   UXTX
X	SXTX

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.10 ADDS (immediate)

Add (immediate), setting flags.

This instruction is used by the alias C<sub>MN</sub> (immediate).

### Syntax

ADDS *Wd*, *Wn/WSP*, #*imm{, shift}* ; 32-bit

ADDS *Xd*, *Xn/SP*, #*imm{, shift}* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn/WSP*

Is the 32-bit name of the source general-purpose register or stack pointer.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn/SP*

Is the 64-bit name of the source general-purpose register or stack pointer.

*imm*

Is an unsigned immediate, in the range 0 to 4095.

*shift*

Is the optional left shift to apply to the immediate, defaulting to LSL #0, and can be either LSL #0 or LSL #12.

### Operation

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.

*Rd* = *Rn* + *shift(imm)*, where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.11 ADDS (shifted register)

Add (shifted register), setting flags.

This instruction is used by the alias C<sub>MN</sub> (shifted register).

### Syntax

ADDS *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

ADDS *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

### Operation

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.

*Rd* = *Rn* + *shift(Rm, amount)*, where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.12 ADR

Form PC-relative address.

### Syntax

*ADR* *Xd*, *Label*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Label*

Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range  $\pm 1\text{MB}$ .

### Usage

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.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.13 ADRP

Form PC-relative address to 4KB page.

### Syntax

*ADRP* *Xd, Label*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*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  $\pm 4\text{GB}$ .

### Usage

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.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.14 AND (immediate)

Bitwise AND (immediate).

### Syntax

AND *Wd/WSP*, *Wn*, #*imm* ; 32-bit

AND *Xd/SP*, *Xn*, #*imm* ; 64-bit

Where:

*Wd/WSP*

Is the 32-bit name of the destination general-purpose register or stack pointer.

*Wn*

Is the 32-bit name of the general-purpose source register.

*imm*

The bitmask immediate.

*Xd/SP*

Is the 64-bit name of the destination general-purpose register or stack pointer.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

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

*Rd* = *Rn* AND *imm*, where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.15 AND (shifted register)

Bitwise AND (shifted register).

### Syntax

AND *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

AND *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

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.

*Rd* = *Rn* AND *shift(Rm, amount)*, where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.16 ANDS (immediate)

Bitwise AND (immediate), setting flags.

This instruction is used by the alias TST (immediate).

### Syntax

ANDS *Wd*, *Wn*, #*imm* ; 32-bit

ANDS *Xd*, *Xn*, #*imm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*imm*

The bitmask immediate.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

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.

*Rd* = *Rn* AND *imm*, where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.17 ANDS (shifted register)

Bitwise AND (shifted register), setting flags.

This instruction is used by the alias TST (shifted register).

### Syntax

ANDS *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

ANDS *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

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.

*Rd* = *Rn* AND *shift(Rm, amount)*, where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.18 ASR (register)

Arithmetic Shift Right (register).

This instruction is an alias of ASRV.

The equivalent instruction is ASRV *Wd*, *Wn*, *Wm*.

### Syntax

ASR *Wd*, *Wn*, *Wm* ; 32-bit

ASR *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

### Operation

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

*Rd* = ASR(*Rn*, *Rm*), where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order on page D2-662](#)

## D2.19 ASR (immediate)

Arithmetic Shift Right (immediate).

This instruction is an alias of SBFM.

The equivalent instruction is SBFM *Wd*, *Wn*, #*shift*, #31.

### Syntax

ASR *Wd*, *Wn*, #*shift* ; 32-bit

ASR *Xd*, *Xn*, #*shift* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*shift*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

*Rd* = ASR(*Rn*, *shift*), where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.20 ASRV

Arithmetic Shift Right Variable.

This instruction is used by the alias ASR (register).

### Syntax

ASRV *Wd*, *Wn*, *Wm* ; 32-bit

ASRV *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

### Operation

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

*Rd* = ASR(*Rn*, *Rm*), where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.21 AT

Address Translate.

This instruction is an alias of SYS.

The equivalent instruction is SYS #op1, C7, Cm, #op2, Xt.

### Syntax

AT at\_op, Xt

Where:

**at\_op**

Is an AT instruction name, as listed for the AT system instruction group, and can be one of the values shown in Usage.

**op1**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Cm**

Is a name Cm, with m in the range 0 to 15.

**op2**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Xt**

Is the 64-bit name of the general-purpose source register.

### Usage

Address Translate. For more information, see *A64 system instructions for address translation* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

The following table shows the valid specifier combinations:

**Table D2-4 SYS parameter values corresponding to AT operations**

at_op	op1	Cm	op2
S12E0R	4	0	6
S12E0W	4	0	7
S12E1R	4	0	4
S12E1W	4	0	5
S1E0R	0	0	2
S1E0W	0	0	3
S1E1R	0	0	0
S1E1RP	0	1	0
S1E1W	0	0	1
S1E1WP	0	1	1
S1E2R	4	0	0
S1E2W	4	0	1
S1E3R	6	0	0
S1E3W	6	0	1

**Related reference**

*D2.1 A64 instructions in alphabetical order* on page D2-662

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D2.22 AUTDA, AUTDZA

Authenticate Data address, using key A.

### Syntax

AUTDA  $Xd$ ,  $Xn/SP$  ; AUTDA general registers

AUTDZA  $Xd$  ; AUTDZA general registers

Where:

$Xn/SP$

Is the 64-bit name of the general-purpose source register or stack pointer.

$Xd$

Is the 64-bit name of the general-purpose destination register.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by  $Xd$ .

The modifier is:

- In the general-purpose register or stack pointer that is specified by  $Xn/SP$  for AUTDA.
- The value zero, for AUTDZA.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.23 AUTDB, AUTDZB

Authenticate Data address, using key B.

### Syntax

AUTDB  $Xd$ ,  $Xn/SP$  ; AUTDB general registers

AUTDZB  $Xd$  ; AUTDZB general registers

Where:

$Xn/SP$

Is the 64-bit name of the general-purpose source register or stack pointer.

$Xd$

Is the 64-bit name of the general-purpose destination register.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by  $Xd$ .

The modifier is:

- In the general-purpose register or stack pointer that is specified by  $Xn/SP$  for AUTDB.
- The value zero, for AUTDZB.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.24 AUTIA, AUTIZA, AUTIA1716, AUTIASP, AUTIAZ

Authenticate Instruction address, using key A.

### Syntax

AUTIA *Xd*, *Xn/SP*

AUTIZA *Xd*

AUTIA1716

AUTIASP

AUTIAZ

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn/SP*

Is the 64-bit name of the general-purpose source register or stack pointer.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by *Xd* for AUTIA and AUTIZA.
- In X17, for AUTIA1716.
- In X30, for AUTIASP and AUTIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn/SP* for AUTIA.
- The value zero, for AUTIZA and AUTIAZ.
- In X16, for AUTIA1716.
- In SP, for AUTIASP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.25 AUTIB, AUTIZB, AUTIB1716, AUTIBSP, AUTIBZ

Authenticate Instruction address, using key B.

### Syntax

AUTIB *Xd*, *Xn/SP*

AUTIZB *Xd*

AUTIB1716

AUTIBSP

AUTIBZ

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn/SP*

Is the 64-bit name of the general-purpose source register or stack pointer.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by *Xd* for AUTIB and AUTIZB.
- In X17, for AUTIB1716.
- In X30, for AUTIBSP and AUTIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn/SP* for AUTIB.
- The value zero, for AUTIZB and AUTIBZ.
- In X16, for AUTIB1716.
- In SP, for AUTIBSP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.26 AXFlag

Convert floating-point condition flags from Arm to external format.

### Syntax

AXFlag

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Convert floating-point condition flags from Arm to external format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an Arm floating-point scalar compare instruction to an alternative representation required by some software.

### *Related reference*

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.27 B.cond

Branch conditionally.

### Syntax

*B.cond Label*

Where:

*cond*

Is one of the standard conditions.

*Label*

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range  $\pm 1\text{MB}$ .

### Usage

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

#### *Related reference*

*Condition code suffixes and related flags*

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.28 B

Branch.

### Syntax

B *Label*

Where:

*Label*

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range  $\pm 128\text{MB}$ . The branch can be forward or backward within 128MB.

### Usage

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.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.29 BFC

Bitfield Clear, leaving other bits unchanged.

This instruction is an alias of BFM.

The equivalent instruction is BFM *Wd*, WZR, #(-*Lsb* MOD 32), #(width-1).

### Syntax

BFC *Wd*, #*Lsb*, #*width* ; 32-bit

BFC *Xd*, #*Lsb*, #*width* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Lsb*

Depends on the instruction variant:

#### 32-bit general registers

Is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

#### 64-bit general registers

Is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

*width*

Depends on the instruction variant:

#### 32-bit general registers

Is the width of the bitfield, in the range 1 to 32-*Lsb*.

#### 64-bit general registers

Is the width of the bitfield, in the range 1 to 64-*Lsb*.

*Xd*

Is the 64-bit name of the general-purpose destination register.

### Architectures supported

Supported in the Armv8.2 architecture and later.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.30 BFI

Bitfield Insert.

This instruction is an alias of BFM.

The equivalent instruction is BFM *Wd*, *Wn*, #(−*Lsb* MOD 32), #(width−1).

### Syntax

BFI *Wd*, *Wn*, #*Lsb*, #*width* ; 32-bit

BFI *Xd*, *Xn*, #*Lsb*, #*width* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Lsb*

Depends on the instruction variant:

#### 32-bit general registers

Is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

#### 64-bit general registers

Is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

*width*

Depends on the instruction variant:

#### 32-bit general registers

Is the width of the bitfield, in the range 1 to 32−*Lsb*.

#### 64-bit general registers

Is the width of the bitfield, in the range 1 to 64−*Lsb*.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

Bitfield Insert 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.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.31 BFM

Bitfield Move.

This instruction is used by the aliases:

- BFC.
- BFI.
- BFXIL.

### Syntax

BFM *Wd*, *Wn*, #<immr>, #<imms> ; 32-bit

BFM *Xd*, *Xn*, #<immr>, #<imms> ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

<immr>

Depends on the instruction variant:

#### 32-bit general registers

Is the right rotate amount, in the range 0 to 31.

#### 64-bit general registers

Is the right rotate amount, in the range 0 to 63.

<imms>

Depends on the instruction variant:

#### 32-bit general registers

Is the leftmost bit number to be moved from the source, in the range 0 to 31.

#### 64-bit general registers

Is the leftmost bit number to be moved from the source, in the range 0 to 63.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

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.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.32 BFXIL

Bitfield extract and insert at low end.

This instruction is an alias of **BFM**.

The equivalent instruction is **BFM *Wd*, *Wn*, #*Lsb*, #(*Lsb+width*-1)**.

### Syntax

**BFXIL *Wd*, *Wn*, #*Lsb*, #*width* ; 32-bit**

**BFXIL *Xd*, *Xn*, #*Lsb*, #*width* ; 64-bit**

Where:

***Wd***

Is the 32-bit name of the general-purpose destination register.

***Wn***

Is the 32-bit name of the general-purpose source register.

***Lsb***

Depends on the instruction variant:

#### 32-bit general registers

Is the bit number of the lsb of the source bitfield, in the range 0 to 31.

#### 64-bit general registers

Is the bit number of the lsb of the source bitfield, in the range 0 to 63.

***width***

Depends on the instruction variant:

#### 32-bit general registers

Is the width of the bitfield, in the range 1 to 32-*Lsb*.

#### 64-bit general registers

Is the width of the bitfield, in the range 1 to 64-*Lsb*.

***Xd***

Is the 64-bit name of the general-purpose destination register.

***Xn***

Is the 64-bit name of the general-purpose source register.

### Usage

Bitfield extract and insert at low end copies any number of low-order bits from a source register into the same number of adjacent bits at the low end in the destination register, leaving other bits unchanged.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.33 BIC (shifted register)

Bitwise Bit Clear (shifted register).

### Syntax

BIC *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

BIC *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

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.

*Rd* = *Rn* AND NOT *shift(Rm, amount)*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.34 BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags.

### Syntax

BICS *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

BICS *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

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.

*Rd* = *Rn* AND NOT *shift(Rm, amount)*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.35 BL

Branch with Link.

### Syntax

`BL Label`

Where:

*Label*

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range  $\pm 128\text{MB}$ . The branch can be forward or backward within 128MB.

### Usage

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.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.36 BLR

Branch with Link to Register.

### Syntax

BLR *Xn*

Where:

*Xn*

Is the 64-bit name of the general-purpose register holding the address to be branched to.

### Usage

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.37 BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication.

### Syntax

BLRAA  $Xn$ ,  $Xm/SP$  ; BLRAA general registers

BLRAAZ  $Xn$  ; BLRAAZ general registers

BLRAB  $Xn$ ,  $Xm/SP$  ; BLRAB general registers

BLRABZ  $Xn$  ; BLRABZ general registers

Where:

$Xn$

Is the 64-bit name of the general-purpose register holding the address to be branched to.

$Xm/SP$

Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by  $Xn$ , using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:

- In the general-purpose register or stack pointer that is specified by  $Xm/SP$  for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.38 BR

Branch to Register.

### Syntax

BR *Xn*

Where:

*Xn*

Is the 64-bit name of the general-purpose register holding the address to be branched to.

### Usage

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

#### *Related reference*

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.39 BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication.

### Syntax

```
BRAA Xn, Xm/SP ; BRAA general registers  
BRAAZ Xn ; BRAAZ general registers  
BRAB Xn, Xm/SP ; BRAB general registers  
BRABZ Xn ; BRABZ general registers
```

Where:

*Xn*

Is the 64-bit name of the general-purpose register holding the address to be branched to.

*Xm/SP*

Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier.

### Architectures supported

Supported in the Armv8.3-A architecture and later.

### Usage

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by *Xn*, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xm/SP* for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.40 BRK

Breakpoint instruction.

### Syntax

BRK #*imm*

Where:

*imm*

Is a 16-bit unsigned immediate, in the range 0 to 65535.

### Usage

Breakpoint instruction generates a Breakpoint Instruction exception. The PE records the exception in ESR\_ELx, using the EC value 0x3c, and captures the value of the immediate argument in ESR\_ELx.ISS.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.41 BTI

Branch Target Identification.

### Syntax

`BTI {<targets>}`

Where:

#### <targets>

Is the type of indirection, and can be one of:

c

Branch Target Identification for function calls. Checks that the two bits, PSTATE.BTYPE, match the value set by BLR instructions. Instruction faults on mismatch.

j

Branch Target Identification for jumps. Checks that the two bits, PSTATE.BTYPE, match the value set by BR instructions.

jc

Branch Target Identification for function calls or jumps. Checks that the two bits, PSTATE.BTYPE, match either the value set by BLR or value set by BR instructions.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Branch Target Identification. A BTI instruction is used to guard against the execution of instructions that are not the intended target of a branch.

Outside of a guarded memory region, a BTI instruction executes as a NOP. In a guarded memory region with PSTATE.BTYPE != 0b00, a BTI instruction compatible with the current value of PSTATE.BTYPE does not generate a Branch Target Exception and allows execution of subsequent instructions within the memory region.

The operand <targets> passed to a BTI instruction determines the values of PSTATE.BTYPE which the BTI instruction is compatible with.

---

#### Note

---

In a guarded memory region, with PSTATE.BTYPE != 0b00, all instructions generate a Branch Target Exception, other than BRK, BTI, HLT, PACIASP, and PACIBSP, which might not. See the individual instructions for details.

---

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.42 CBNZ

Compare and Branch on Nonzero.

### Syntax

`CBNZ Wt, Label ; 32-bit`

`CBNZ Xt, Label ; 64-bit`

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be tested.

*Xt*

Is the 64-bit name of the general-purpose register to be tested.

*Label*

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range  $\pm 1\text{MB}$ .

### Usage

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.

### *Related reference*

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.43 CBZ

Compare and Branch on Zero.

### Syntax

`CBZ Wt, Label ; 32-bit`

`CBZ Xt, Label ; 64-bit`

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be tested.

*Xt*

Is the 64-bit name of the general-purpose register to be tested.

*Label*

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range  $\pm 1\text{MB}$ .

### Usage

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.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.44 CCMN (immediate)

Conditional Compare Negative (immediate).

### Syntax

CCMN *Wn*, #*imm*, #*nzcv*, *cond* ; 32-bit

CCMN *Xn*, #*imm*, #*nzcv*, *cond* ; 64-bit

Where:

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*imm*

Is a five bit unsigned immediate.

*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.

*cond*

Is one of the standard conditions.

### Operation

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.

flags = if *cond* then compare(*Rn*, #-*imm*) else #*nzcv*, where *R* is either *W* or *X*.

### Related reference

[D1.8 Condition code suffixes and related flags on page D1-655](#)

[D2.1 A64 instructions in alphabetical order on page D2-662](#)

## D2.45 CCMN (register)

Conditional Compare Negative (register).

### Syntax

CCMN *Wn*, *Wm*, #*nzcv*, *cond* ; 32-bit

CCMN *Xn*, *Xm*, #*nzcv*, *cond* ; 64-bit

Where:

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*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.

*cond*

Is one of the standard conditions.

### Operation

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.

flags = if *cond* then compare(*Rn*, -*Rm*) else #*nzcv*, where *R* is either *W* or *X*.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.46 CCMP (immediate)

Conditional Compare (immediate).

### Syntax

CCMP *Rn*, #*imm*, #*nzcv*, *cond* ; 32-bit

CCMP *Xn*, #*imm*, #*nzcv*, *cond* ; 64-bit

Where:

*Rn*

Is the 32-bit name of the first general-purpose source register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*imm*

Is a five bit unsigned immediate.

*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.

*cond*

Is one of the standard conditions.

### Operation

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.

*flags* = if *cond* then compare(*Rn*, #*imm*) else #*nzcv*, where *R* is either W or X.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.47 CCMP (register)

Conditional Compare (register).

### Syntax

CCMP *Rn*, *Rm*, #*nzcv*, *cond* ; 32-bit

CCMP *Xn*, *Xm*, #*nzcv*, *cond* ; 64-bit

Where:

*Rn*

Is the 32-bit name of the first general-purpose source register.

*Rm*

Is the 32-bit name of the second general-purpose source register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*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.

*cond*

Is one of the standard conditions.

### Operation

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.

flags = if *cond* then compare(*Rn*, *Rm*) else #*nzcv*, where *R* is either *W* or *X*.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.48 CINC

Conditional Increment.

This instruction is an alias of CSINC.

The equivalent instruction is CSINC *Wd*, *Wn*, *Wn*, invert(*cond*).

### Syntax

CINC *Wd*, *Wn*, *cond* ; 32-bit

CINC *Xd*, *Xn*, *cond* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

*cond*

Is one of the standard conditions, excluding AL and NV.

### Operation

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

*Rd* = if *cond* then *Rn+1* else *Rn*, where *R* is either W or X.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.49 CINV

Conditional Invert.

This instruction is an alias of CSINV.

The equivalent instruction is CSINV *Wd*, *Wn*, *Wn*, invert(*cond*).

### Syntax

CINV *Wd*, *Wn*, *cond* ; 32-bit

CINV *Xd*, *Xn*, *cond* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

*cond*

Is one of the standard conditions, excluding AL and NV.

### Operation

Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

*Rd* = if *cond* then NOT(*Rn*) else *Rn*, where *R* is either *W* or *X*.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.50 CLREX

Clear Exclusive.

### Syntax

CLREX {#*imm*}

Where:

*imm*

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15.

### Usage

Clear Exclusive clears the local monitor of the executing PE.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.51 CLS

Count leading sign bits.

### Syntax

CLS *Wd, Wn* ; 32-bit

CLS *Xd, Xn* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

*Rd* = CLS(*Rn*), where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.52 CLZ

Count leading zero bits.

### Syntax

CLZ *Wd*, *Wn* ; 32-bit

CLZ *Xd*, *Xn* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

*Rd* = CLZ(*Rn*), where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.53 CMN (extended register)

Compare Negative (extended register).

This instruction is an alias of ADDS (extended register).

The equivalent instruction is ADDS WZR, *Wn/WSP*, *Wm{*, *extend {#amount}}**}*.

### Syntax

CMN *Wn/WSP*, *Wm{*, *extend {#amount}}**}* ; 32-bit

CMN *Xn/SP*, *Rm{*, *extend {#amount}}**}* ; 64-bit

Where:

*Wn/WSP*

Is the 32-bit name of the first source general-purpose register or stack pointer.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*extend*

Is the extension to be applied to the second source operand:

#### 32-bit general registers

Can be one of UXTB, UXTH, LSL | UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

#### 64-bit general registers

Can be one of UXTB, UXTH, UXTW, LSL | UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

*Xn/SP*

Is the 64-bit name of the first source general-purpose register or stack pointer.

*R*

Is a width specifier, and can be either W or X.

*m*

Is the number [0-30] of the second general-purpose source register or the name ZR (31).

*amount*

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. 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

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. 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, and discards the result.

*Rn* + LSL(*extend(Rm)*, *amount*), where *R* is either W or X.

## Usage

**Table D2-5 CMN (64-bit general registers) specifier combinations**

<i>R</i>	<i>extend</i>
W	SXTB
W	SXTH
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL   UXTX
X	SXTX

### *Related reference*

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.54 CMN (immediate)

Compare Negative (immediate).

This instruction is an alias of ADDS (immediate).

The equivalent instruction is ADDS WZR, *Wn/WSP*, #*imm* {, *shift*}.

### Syntax

CMN *Wn/WSP*, #*imm*{, *shift*} ; 32-bit

CMN *Xn/SP*, #*imm*{, *shift*} ; 64-bit

Where:

*Wn/WSP*

Is the 32-bit name of the source general-purpose register or stack pointer.

*Xn/SP*

Is the 64-bit name of the source general-purpose register or stack pointer.

*imm*

Is an unsigned immediate, in the range 0 to 4095.

*shift*

Is the optional left shift to apply to the immediate, defaulting to LSL #0, and can be either LSL #0 or LSL #12.

### Operation

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

*Rn* + *shift*(*imm*), where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.55 CMN (shifted register)

Compare Negative (shifted register).

This instruction is an alias of ADDS (shifted register).

The equivalent instruction is ADDS WZR, *Rn*, *Rm* {, *shift #amount*}.

### Syntax

CMN *Rn*, *Rm*{, *shift #amount*} ; 32-bit

CMN *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Rn*

Is the 32-bit name of the first general-purpose source register.

*Rm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

### Operation

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

*Rn* + *shift(Rm, amount)*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.56 CMP (extended register)

Compare (extended register).

This instruction is an alias of SUBS (extended register).

The equivalent instruction is SUBS WZR, *Wn/WSP*, *Wm{*, *extend {#amount}}**}*.

### Syntax

CMP *Wn/WSP*, *Wm{*, *extend {#amount}}**}* ; 32-bit

CMP *Xn/SP*, *Rm{*, *extend {#amount}}**}* ; 64-bit

Where:

*Wn/WSP*

Is the 32-bit name of the first source general-purpose register or stack pointer.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*extend*

Is the extension to be applied to the second source operand:

#### 32-bit general registers

Can be one of UXTB, UXTH, LSL | UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

#### 64-bit general registers

Can be one of UXTB, UXTH, UXTW, LSL | UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

*Xn/SP*

Is the 64-bit name of the first source general-purpose register or stack pointer.

*R*

Is a width specifier, and can be either W or X.

*m*

Is the number [0-30] of the second general-purpose source register or the name ZR (31).

*amount*

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. 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

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. 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, and discards the result.

*Rn* - LSL(*extend(Rm)*, *amount*), where *R* is either W or X.

## Usage

**Table D2-6 CMP (64-bit general registers) specifier combinations**

<i>R</i>	<i>extend</i>
W	SXTB
W	SXTH
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL   UXTX
X	SXTX

### *Related reference*

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.57 CMP (immediate)

Compare (immediate).

This instruction is an alias of SUBS (immediate).

The equivalent instruction is SUBS WZR, *Wn/WSP*, #imm {, shift}.

### Syntax

CMP *Wn/WSP*, #imm{, shift} ; 32-bit

CMP *Xn/SP*, #imm{, shift} ; 64-bit

Where:

*Wn/WSP*

Is the 32-bit name of the source general-purpose register or stack pointer.

*Xn/SP*

Is the 64-bit name of the source general-purpose register or stack pointer.

*imm*

Is an unsigned immediate, in the range 0 to 4095.

*shift*

Is the optional left shift to apply to the immediate, defaulting to LSL #0, and can be either LSL #0 or LSL #12.

### Operation

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

*Rn - shift(imm)*, where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.58 CMP (shifted register)

Compare (shifted register).

This instruction is an alias of SUBS (shifted register).

The equivalent instruction is SUBS WZR, *Rn*, *Rm* {, *shift #amount*}.

### Syntax

CMP *Rn*, *Rm*{, *shift #amount*} ; 32-bit

CMP *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Rn*

Is the 32-bit name of the first general-purpose source register.

*Rm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

### Operation

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

*Rn* - *shift(Rm,amount)*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.59 CMPP

Compare with Tag.

This instruction is an alias of SUBPS.

The equivalent instruction is SUBPS XZR,  $Xn/SP$ ,  $Xm/SP$ .

### Syntax

CMPP  $Xn/SP$ ,  $Xm/SP$

Where:

$Xn/SP$

Is the 64-bit name of the first source general-purpose register or stack pointer.

$Xm/SP$

Is the 64-bit name of the second general-purpose source register or stack pointer.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

#### Related reference

[D2.164 SUBPS on page D2-840](#)

[D2.1 A64 instructions in alphabetical order on page D2-662](#)

## D2.60 CNEG

Conditional Negate.

This instruction is an alias of CSNEG.

The equivalent instruction is CSNEG *Wd*, *Wn*, *Wn*, *invert(cond)*.

### Syntax

CNEG *Wd*, *Wn*, *cond* ; 32-bit

CNEG *Xd*, *Xn*, *cond* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

*cond*

Is one of the standard conditions, excluding AL and NV.

### Operation

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

*Rd* = if *cond* then *-Rn* else *Rn*, where *R* is either *W* or *X*.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.61 CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register.

### Syntax

```
CRC32B Wd, Wn, Wm ; Wd = CRC32(Wn, Rm[<7:0>])  
CRC32H Wd, Wn, Wm ; Wd = CRC32(Wn, Rm[<15:0>])  
CRC32W Wd, Wn, Wm ; Wd = CRC32(Wn, Rm[<31:0>])  
CRC32X Wd, Wn, Xm ; Wd = CRC32(Wn, Rm[<63:0>])
```

Where:

*Wm*

Is the 32-bit name of the general-purpose data source register.

*Xm*

Is the 64-bit name of the general-purpose data source register.

*Wd*

Is the 32-bit name of the general-purpose accumulator output register.

*Wn*

Is the 32-bit name of the general-purpose accumulator input register.

### Operation

This instruction 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.

#### Note

ID\_AA64ISAR0\_EL1.CRC32 indicates whether this instruction is supported. See *ID\_AA64ISAR0\_EL1* in the [Arm® Architecture Reference Manual Armv8, for Arm®v8-A architecture profile](#).

Wd = CRC32(Wn, Rm<n:0>) // n = 7, 15, 31, 63.

### Architectures supported

Supported in architecture Armv8.1 and later. Optionally supported in Armv8-A.

#### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

#### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.62 CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32C checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register.

### Syntax

```
CRC32CB Wd, Wn, Wm ; Wd = CRC32C(Wn, Rm[<7:0>])  
CRC32CH Wd, Wn, Wm ; Wd = CRC32C(Wn, Rm[<15:0>])  
CRC32CW Wd, Wn, Wm ; Wd = CRC32C(Wn, Rm[<31:0>])  
CRC32CX Wd, Wn, Xm ; Wd = CRC32C(Wn, Rm[<63:0>])
```

Where:

*Wm*

Is the 32-bit name of the general-purpose data source register.

*Xm*

Is the 64-bit name of the general-purpose data source register.

*Wd*

Is the 32-bit name of the general-purpose accumulator output register.

*Wn*

Is the 32-bit name of the general-purpose accumulator input register.

### Operation

This instruction 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.

#### Note

ID\_AA64ISAR0\_EL1.CRC32 indicates whether this instruction is supported. See *ID\_AA64ISAR0\_EL1* in the [Arm® Architecture Reference Manual Armv8, for Arm®v8-A architecture profile](#).

Wd = CRC32C(Wn, Rm<n:0>) // n = 7, 15, 31, 63.

### Architectures supported

Supported in architecture Armv8.1 and later. Optionally supported in Armv8-A.

#### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

#### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.63 CSDB

Consumption of Speculative Data Barrier.

### Syntax

CSDB

### Usage

Consumption of Speculative Data Barrier is a memory barrier that controls Speculative execution and data value prediction. Arm Compiler supports the mitigation of the Variant 1 mechanism that is described in the whitepaper at [Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism](#).

The CSDB instruction allows Speculative execution of:

- Branch instructions.
- Instructions that are not a result of data value predictions.
- Instructions that are the result of PSTATE.{N,Z,C,V} predictions from conditional branch instructions.
- Instructions that are not a result of predictions of SVE prediction state for any SVE instructions.

The CSDB instruction prevents Speculative execution of:

- Non-branch instructions.
- Instructions that are the result of data value predictions.
- Instructions that are the result of PSTATE.{N,Z,C,V} predictions from instructions other than conditional branch instructions.
- Instructions that are the result of predictions of SVE prediction state for any SVE instructions.

### Examples

The following example shows a code sequence that could result in the processor loading data from an untrusted location that is provided by a user as the result of Speculative execution of instructions:

```
CMP X0, X1
BGE out_of_range
LDRB W4, [X5, X0]    ; load data from list A
                      ; speculative execution of this instruction
                      ; must be prevented
AND X4, X4, #1
LSL X4, X4, #8
ADD X4, X4, #0x200
CMP X4, X6
BGE out_of_range
LDRB X7, [X8, X4]    ; load data from list B
out_of_range
```

In this example:

- There are two list objects A and B.
- A contains a list of values that are used to calculate offsets from which data can be loaded from B.
- X1 is the length of A.
- X5 is the base address of A.
- X6 is the length of B.
- X8 is the base address of B.
- X0 is an untrusted offset that is provided by a user, and is used to load an element from A.

When X0 is greater-than or equal-to the length of A, it is outside the address range of A. Therefore, the first branch instruction BGE out\_of\_range is taken, and instructions LDRB W4, [X5, X0] through LDRB X7, [X8, X4] are skipped.

Without a CSDB instruction, these skipped instructions can still be Speculatively executed:

- If X0 is maliciously set to an incorrect value, then data can be loaded into W4 from an address outside the address range of A.
- Data can be loaded into X7 from an address outside the address range of B.

To mitigate against these untrusted accesses, add a pair of CSEL and CSDB instructions between the BGE `out_of_range` and LDRB `W4, [X5, X0]` instructions as follows:

```
CMP X0, X1
BGE out_of_range

CSEL X0, XZR, X0, GE      ; conditionally clears the untrusted
                           ; offset provided by the user so that
                           ; it cannot affect any other code

CSDB                      ; new barrier instruction

LDRB W4, [X5, X0]          ; load data from list A
                           ; speculative execution of this instruction
                           ; is prevented

AND X4, X4, #1
LSL X4, X4, #8
ADD X4, X4, #0x200
CMP X4, X6
BGE out_of_range
LDRB X7, [X8, X4]          ; load data from list B
out_of_range
```

#### **Related reference**

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

[D2.64 CSEL](#) on page D2-737

#### **Related information**

[Arm Processor Security Update](#)

[Compiler support for mitigations](#)

## D2.64 CSEL

Conditional Select.

### Syntax

CSEL *Wd*, *Wn*, *Wm*, *cond* ; 32-bit

CSEL *Xd*, *Xn*, *Xm*, *cond* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*cond*

Is one of the standard conditions.

### Operation

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.

*Rd* = if *cond* then *Rn* else *Rm*, where *R* is either *W* or *X*.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.65 CSET

Conditional Set.

This instruction is an alias of CSINC.

The equivalent instruction is CSINC *Wd*, WZR, WZR, invert(*cond*).

### Syntax

CSET *Wd*, *cond* ; 32-bit

CSET *Xd*, *cond* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*cond*

Is one of the standard conditions, excluding AL and NV.

### Operation

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

Rd = if *cond* then 1 else 0, where R is either W or X.

### Related reference

[D1.8 Condition code suffixes and related flags on page D1-655](#)

[D2.1 A64 instructions in alphabetical order on page D2-662](#)

## D2.66 CSETM

Conditional Set Mask.

This instruction is an alias of CSINV.

The equivalent instruction is CSINV *Wd*, WZR, WZR, invert(*cond*).

### Syntax

CSETM *Wd*, *cond* ; 32-bit

CSETM *Xd*, *cond* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*cond*

Is one of the standard conditions, excluding AL and NV.

### Operation

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

*Rd* = if *cond* then -1 else 0, where *R* is either W or X.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.67 CSINC

Conditional Select Increment.

This instruction is used by the aliases:

- CINC.
- CSET.

### Syntax

CSINC *Wd*, *Wn*, *Wm*, *cond* ; 32-bit

CSINC *Xd*, *Xn*, *Xm*, *cond* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*cond*

Is one of the standard conditions.

### Operation

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.

*Rd* = if *cond* then *Rn* else (*Rm* + 1), where *R* is either *W* or *X*.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.68 CSINV

Conditional Select Invert.

This instruction is used by the aliases:

- CINV.
- CSETM.

### Syntax

`CSINV Wd, Wn, Wm, cond ; 32-bit`

`CSINV Xd, Xn, Xm, cond ; 64-bit`

Where:

`Wd`

Is the 32-bit name of the general-purpose destination register.

`Wn`

Is the 32-bit name of the first general-purpose source register.

`Wm`

Is the 32-bit name of the second general-purpose source register.

`Xd`

Is the 64-bit name of the general-purpose destination register.

`Xn`

Is the 64-bit name of the first general-purpose source register.

`Xm`

Is the 64-bit name of the second general-purpose source register.

`cond`

Is one of the standard conditions.

### Operation

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.

`Rd = if cond then Rn else NOT (Rm)`, where `R` is either `W` or `X`.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.69 CSNEG

Conditional Select Negation.

This instruction is used by the alias CNEG.

### Syntax

CSNEG *Wd*, *Wn*, *Wm*, *cond* ; 32-bit

CSNEG *Xd*, *Xn*, *Xm*, *cond* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*cond*

Is one of the standard conditions.

### Operation

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.

*Rd* = if *cond* then *Rn* else *-Rm*, where *R* is either *W* or *X*.

### Related reference

[D1.8 Condition code suffixes and related flags on page D1-655](#)

[D2.1 A64 instructions in alphabetical order on page D2-662](#)

## D2.70 DC

Data Cache operation.

This instruction is an alias of **SYS**.

The equivalent instruction is **SYS #op1, C7, Cm, #op2, Xt**.

### Syntax

**DC <dc\_op>, Xt**

Where:

**<dc\_op>**

Is a DC instruction name, as listed for the DC system instruction group, and can be one of the values shown in Usage.

**op1**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Cm**

Is a name **Cm**, with **m** in the range 0 to 15.

**op2**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Xt**

Is the 64-bit name of the general-purpose source register.

### Usage

Data Cache operation. For more information, see *A64 system instructions for cache maintenance* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The following table shows the valid specifier combinations:

**Table D2-7 SYS parameter values corresponding to DC operations**

<b>&lt;dc_op&gt;</b>	<b>op1</b>	<b>Cm</b>	<b>op2</b>
CISW	0	14	2
CIVAC	3	14	1
CSW	0	10	2
CVAC	3	10	1
CVAP	3	12	1
CVAU	3	11	1
ISW	0	6	2
IVAC	0	6	1
ZVA	3	4	1

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D2.71 DCPS1

Debug Change PE State to EL1.

### Syntax

`DCPS1 {#imm}`

Where:

`imm`

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0.

### Usage

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 `SCTRLR_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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D2.72 DCPS2

Debug Change PE State to EL2.

### Syntax

DCPS2 {#*imm*}

Where:

*imm*

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0.

### Usage

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 *SCTRLR\_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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.73 DCPS3

Debug Change PE State to EL3.

### Syntax

DCPS3 {#*imm*}

Where:

*imm*

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0.

### Usage

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 *SCTRLR\_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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.74 DMB

Data Memory Barrier.

### Syntax

`DMB option|#imm`

Where:

#### *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 in both Group A and Group B. This option is referred to as the full system DMB.

##### **ST**

Full system is the required shareability domain, writes are the required access type in both Group A and Group B.

##### **LD**

Full system is the required shareability domain, reads are the required access type in Group A, and reads and writes are the required access types in Group B.

##### **ISH**

Inner Shareable is the required shareability domain, reads and writes are the required access types in both Group A and Group B.

##### **ISHST**

Inner Shareable is the required shareability domain, writes are the required access type in both Group A and Group B.

##### **ISHLD**

Inner Shareable is the required shareability domain, reads are the required access type in Group A, and reads and writes are the required access types in Group B.

##### **NSH**

Non-shareable is the required shareability domain, reads and writes are the required access types in both Group A and Group B.

##### **NSHST**

Non-shareable is the required shareability domain, writes are the required access type in both Group A and Group B.

##### **NSHLD**

Non-shareable is the required shareability domain, reads are the required access type in Group A, and reads and writes are the required access types in Group B.

##### **OSH**

Outer Shareable is the required shareability domain, reads and writes are the required access types in both Group A and Group B.

##### **OSHST**

Outer Shareable is the required shareability domain, writes are the required access type in both Group A and Group B.

##### **OSHLD**

Outer Shareable is the required shareability domain, reads are the required access type in Group A, and reads and writes are the required access types in Group B.

##### *imm*

Is a 4-bit unsigned immediate, in the range 0 to 15.

### Usage

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

**Related reference**

*D2.1 A64 instructions in alphabetical order* on page D2-662

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D2.75 DRPS

Debug restore process state.

### Syntax

DRPS

#### *Related reference*

[D2.1 A64 instructions in alphabetical order on page D2-662](#)

## D2.76 DSB

Data Synchronization Barrier.

### Syntax

`DSB option|#imm`

Where:

#### `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 in both Group A and Group B. This option is referred to as the full system DMB.

##### `ST`

Full system is the required shareability domain, writes are the required access type in both Group A and Group B.

##### `LD`

Full system is the required shareability domain, reads are the required access type in Group A, and reads and writes are the required access types in Group B.

##### `ISH`

Inner Shareable is the required shareability domain, reads and writes are the required access types in both Group A and Group B.

##### `ISHST`

Inner Shareable is the required shareability domain, writes are the required access type in both Group A and Group B.

##### `ISHLD`

Inner Shareable is the required shareability domain, reads are the required access type in Group A, and reads and writes are the required access types in Group B.

##### `NSH`

Non-shareable is the required shareability domain, reads and writes are the required access types in both Group A and Group B.

##### `NSHST`

Non-shareable is the required shareability domain, writes are the required access type in both Group A and Group B.

##### `NSHLD`

Non-shareable is the required shareability domain, reads are the required access type in Group A, and reads and writes are the required access types in Group B.

##### `OSH`

Outer Shareable is the required shareability domain, reads and writes are the required access types in both Group A and Group B.

##### `OSHST`

Outer Shareable is the required shareability domain, writes are the required access type in both Group A and Group B.

##### `OSHLD`

Outer Shareable is the required shareability domain, reads are the required access type in Group A, and reads and writes are the required access types in Group B.

##### `imm`

Is a 4-bit unsigned immediate, in the range 0 to 15.

### Usage

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

**Related reference**

*D2.1 A64 instructions in alphabetical order* on page D2-662

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D2.77 EON (shifted register)

Bitwise Exclusive OR NOT (shifted register).

### Syntax

EON *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

EON *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

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.

*Rd* = *Rn* EOR NOT *shift(Rm, amount)*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.78 EOR (immediate)

Bitwise Exclusive OR (immediate).

### Syntax

EOR *Wd/WSP*, *Wn*, #*imm* ; 32-bit

EOR *Xd/SP*, *Xn*, #*imm* ; 64-bit

Where:

*Wd/WSP*

Is the 32-bit name of the destination general-purpose register or stack pointer.

*Wn*

Is the 32-bit name of the general-purpose source register.

*imm*

The bitmask immediate.

*Xd/SP*

Is the 64-bit name of the destination general-purpose register or stack pointer.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

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.

*Rd* = *Rn* EOR *imm*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.79 EOR (shifted register)

Bitwise Exclusive OR (shifted register).

### Syntax

EOR *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

EOR *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

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.

*Rd* = *Rn* EOR *shift(Rm, amount)*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.80 ERET

Returns from an exception. It restores the processor state based on SPSR\_EL $n$  and branches to ELR\_EL $n$ , where  $n$  is the current exception level..

### Syntax

ERET

### Usage

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

ERET is UNDEFINED at EL0.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.81 EREAA, ERETAB

Exception Return, with pointer authentication.

### Syntax

EREAA ; EREAA general registers

ERETAB ; ERETAB general registers

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for EREAA, and key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

ERET is UNDEFINED at EL0.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.82 ESB

Error Synchronization Barrier.

### Syntax

ESB

### Architectures supported

Supported in the Armv8.2 architecture and later.

### Usage

Error Synchronization Barrier.

### *Related reference*

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.83 EXTR

Extract register.

This instruction is used by the alias ROR (immediate).

### Syntax

EXTR *Wd*, *Wn*, *Wm*, #*Lsb* ; 32-bit

EXTR *Xd*, *Xn*, *Xm*, #*Lsb* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Lsb*

Depends on the instruction variant:

#### 32-bit general registers

Is the least significant bit position from which to extract, in the range 0 to 31.

#### 64-bit general registers

Is the least significant bit position from which to extract, in the range 0 to 63.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

### Usage

Extract register extracts a register from a pair of registers.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.84 GMI

Tag Mask Insert.

### Syntax

GMI  $Xd$ ,  $Xn/SP$ ,  $Xm$

Where:

$Xd$

Is the 64-bit name of the general-purpose destination register.

$Xn/SP$

Is the 64-bit name of the first source general-purpose register or stack pointer.

$Xm$

Is the 64-bit name of the second general-purpose source register.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.85 HINT

Hint instruction.

### Syntax

```
HINT #imm ; Hints 6 and 7  
HINT #imm ; Hints 8 to 15, and 24 to 127  
HINT #imm ; Hints 17 to 23
```

Where:

*imm*

Is a 7-bit unsigned immediate, in the range 0 to 127, but excludes the following:

- 0** NOP.
- 1** YIELD.
- 2** WFE.
- 3** WFI.
- 4** SEV.
- 5** SEVL.

### Usage

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

The encoding variants described here are unallocated in this revision of the architecture, and behave as a NOP. These encodings might be allocated to other hint functionality in future revisions of the architecture.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.86 HLT

Halt instruction.

### Syntax

`HLT #imm`

Where:

`imm`

Is a 16-bit unsigned immediate, in the range 0 to 65535.

### Usage

Halt instruction generates a Halt Instruction debug event.

#### *Related reference*

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.87 HVC

Hypervisor call to allow OS code to call the Hypervisor. It generates an exception targeting exception level 2 (EL2).

### Syntax

HVC #imm

Where:

imm

Is a 16-bit unsigned immediate, in the range 0 to 65535. This value is made available to the handler in the Exception Syndrome Register.

### Usage

Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- At EL0, and Secure EL1.
- When SCR\_EL3.HCE is set to 0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in ESR\_ELx, using the EC value 0x16, and the value of the immediate argument.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.88 IC

Instruction Cache operation.

This instruction is an alias of **SYS**.

The equivalent instruction is **SYS #op1, C7, Cm, #op2{, Xt}**.

### Syntax

**IC <ic\_op>{, Xt}**

Where:

**<ic\_op>**

Is an IC instruction name, as listed for the IC system instruction pages, and can be one of the values shown in Usage.

**op1**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Cm**

Is a name **Cm**, with **m** in the range 0 to 15.

**op2**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Xt**

Is the 64-bit name of the optional general-purpose source register, defaulting to 31.

### Usage

Instruction Cache operation. For more information, see *A64 system instructions for cache maintenance* in the [Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile](#).

The following table shows the valid specifier combinations:

**Table D2-8 SYS parameter values corresponding to IC operations**

<b>&lt;ic_op&gt;</b>	<b>op1</b>	<b>Cm</b>	<b>op2</b>
IALLU	0	5	0
IALLUIS	0	1	0
IVAU	3	5	1

#### *Related reference*

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

#### *Related information*

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.89 IRG

Insert Random Tag.

### Syntax

IRG  $Xd/SP$ ,  $Xn/SP\{, Xm\}$

Where:

$Xd/SP$

Is the 64-bit name of the destination general-purpose register or stack pointer.

$Xn/SP$

Is the 64-bit name of the first source general-purpose register or stack pointer.

$Xm$

Is the 64-bit name of the second general-purpose source register. Defaults to XZR if absent.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR\_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.90 ISB

Instruction Synchronization Barrier.

### Syntax

`ISB {option|#imm}`

Where:

#### *option*

Specifies an optional limitation on the barrier operation. Values are:

##### `SY`

Full system barrier operation. Can be omitted.

##### `imm`

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15.

### Usage

Instruction Synchronization Barrier flushes the pipeline in the PE, so that all instructions following the `ISB` are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context changing operations executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`. Context changing operations include changing the ASID, TLB maintenance instructions, and all changes to the System registers. In addition, any branches that appear in program order after the `ISB` instruction are written into the branch prediction logic with the context that is visible after the `ISB` instruction. This is needed to ensure correct execution of the instruction stream. For more information, see *Instruction Synchronization Barrier (ISB)* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.91 LDG

Load Allocation Tag.

### Syntax

LDG *Xt*, [*Xn/SP*{, #*simm*}]

Where:

**Xt**

Is the 64-bit name of the general-purpose register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**simm**

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Load Allocation Tag loads an Allocation Tag from a memory address, generates an address with the Logical Address Tag generated from the loaded Allocation Tag, and writes the result to the destination register. The address used for the load is calculated from the source register and an immediate signed offset scaled by the Tag granule.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.92 LDGV

Load Allocation Tag.

### Syntax

LDGV *Xt*, [*Xn/SP*]!

Where:

***Xt***

Is the 64-bit name of the general-purpose register to be transferred.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Load Tag Vector loads an IMPLEMENTATION DEFINED number of Allocation Tags from the naturally aligned array of 16 Allocation Tags which includes a tag whose address is the address in the source register, and writes them to the destination register. Bits of the destination register which do not store a tag are set to 0. The Allocation Tag at the address in the source register is always loaded, and the first source register is updated to the address of the first Allocation Tag at an address higher than the original address that was not loaded.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.93 LSL (register)

Logical Shift Left (register).

This instruction is an alias of LSLV.

The equivalent instruction is LSLV *Wd*, *Wn*, *Wm*.

### Syntax

LSL *Wd*, *Wn*, *Wm* ; 32-bit

LSL *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

### Operation

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

*Rd* = LSL(*Rn*, *Rm*), where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.94 LSL (immediate)

Logical Shift Left (immediate).

This instruction is an alias of UBFM.

The equivalent instruction is UBFM *Wd*, *Wn*, #(−*shift* MOD 32), #(31−*shift*).

### Syntax

LSL *Wd*, *Wn*, #*shift* ; 32-bit

LSL *Xd*, *Xn*, #*shift* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*shift*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

*Rd* = LSL(*Rn*, *shift*), where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.95 LSLV

Logical Shift Left Variable.

This instruction is used by the alias LSL (register).

### Syntax

LSLV *Wd*, *Wn*, *Wm* ; 32-bit

LSLV *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

### Operation

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

$Rd = LSL(Rn, Rm)$ , where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.96 LSR (register)

Logical Shift Right (register).

This instruction is an alias of LSRV.

The equivalent instruction is LSRV *Wd*, *Wn*, *Wm*.

### Syntax

LSR *Wd*, *Wn*, *Wm* ; 32-bit

LSR *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

### Operation

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

*Rd* = LSR(*Rn*, *Rm*), where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.97 LSR (immediate)

Logical Shift Right (immediate).

This instruction is an alias of UBFM.

The equivalent instruction is UBFM *Wd*, *Wn*, #*shift*, #31.

### Syntax

LSR *Wd*, *Wn*, #*shift* ; 32-bit

LSR *Xd*, *Xn*, #*shift* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*shift*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

*Rd* = LSR(*Rn*, *shift*), where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.98 LSRV

Logical Shift Right Variable.

This instruction is used by the alias LSR (register).

### Syntax

LSRV *Wd*, *Wn*, *Wm* ; 32-bit

LSRV *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

### Operation

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

*Rd* = LSR(*Rn*, *Rm*), where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.99 MADD

Multiply-Add.

This instruction is used by the alias **MUL**.

### Syntax

**MADD Wd, Wn, Wm, Wa ; 32-bit**

**MADD Xd, Xn, Xm, Xa ; 64-bit**

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Wn**

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

**Wm**

Is the 32-bit name of the second general-purpose source register holding the multiplier.

**Wa**

Is the 32-bit name of the third general-purpose source register holding the addend.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Xn**

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

**Xm**

Is the 64-bit name of the second general-purpose source register holding the multiplier.

**Xa**

Is the 64-bit name of the third general-purpose source register holding the addend.

### Operation

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

$Rd = Ra + Rn * Rm$ , where  $R$  is either  $W$  or  $X$ .

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.100 MNEG

Multiply-Negate.

This instruction is an alias of MSUB.

The equivalent instruction is MSUB *Wd*, *Wn*, *Wm*, WZR.

### Syntax

MNEG *Wd*, *Wn*, *Wm* ; 32-bit

MNEG *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

*Xm*

Is the 64-bit name of the second general-purpose source register holding the multiplier.

### Operation

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

*Rd* = -(*Rn* \* *Rm*), where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.101 MOV (to or from SP)

Move between register and stack pointer.

This instruction is an alias of ADD (immediate).

The equivalent instruction is ADD *Wd/WSP*, *Wn/WSP*, #0.

### Syntax

MOV *Wd/WSP*, *Wn/WSP* ; 32-bit

MOV *Xd/SP*, *Xn/SP* ; 64-bit

Where:

*Wd/WSP*

Is the 32-bit name of the destination general-purpose register or stack pointer.

*Wn/WSP*

Is the 32-bit name of the source general-purpose register or stack pointer.

*Xd/SP*

Is the 64-bit name of the destination general-purpose register or stack pointer.

*Xn/SP*

Is the 64-bit name of the source general-purpose register or stack pointer.

### Operation

*Rd* = *Rn*, where *R* is either W or X.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.102 MOV (inverted wide immediate)

Move (inverted wide immediate).

This instruction is an alias of **MOVN**.

The equivalent instruction is **MOVN Rd, #imm16, LSL #shift**.

### Syntax

**MOV Rd, #imm ; 32-bit**

**MOV Xd, #imm ; 64-bit**

Where:

**Rd**

Is the 32-bit name of the general-purpose destination register.

**imm**

Depends on the instruction variant:

#### 32-bit general registers

Is a 32-bit immediate.

#### 64-bit general registers

Is a 64-bit immediate.

**Xd**

Is the 64-bit name of the general-purpose destination register.

### Operation

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

**Rd = imm**, where R is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.103 MOV (wide immediate)

Move (wide immediate).

This instruction is an alias of `MOVZ`.

The equivalent instruction is `MOVZ Rd, #imm16, LSL #shift`.

### Syntax

`MOV Rd, #imm ; 32-bit`

`MOV Xd, #imm ; 64-bit`

Where:

**Rd**

Is the 32-bit name of the general-purpose destination register.

**imm**

Depends on the instruction variant:

#### 32-bit general registers

Is a 32-bit immediate.

#### 64-bit general registers

Is a 64-bit immediate.

**Xd**

Is the 64-bit name of the general-purpose destination register.

### Operation

Move (wide immediate) moves a 16-bit immediate value to a register.

$Rd = imm$ , where  $R$  is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.104 MOV (bitmask immediate)

Move (bitmask immediate).

This instruction is an alias of ORR (immediate).

The equivalent instruction is ORR *Wd/WSP*, WZR, #*imm*.

### Syntax

MOV *Wd/WSP*, #*imm* ; 32-bit

MOV *Xd/SP*, #*imm* ; 64-bit

Where:

*Wd/WSP*

Is the 32-bit name of the destination general-purpose register or stack pointer.

*imm*

The bitmask immediate but excluding values which could be encoded by MOVZ or MOVN.

*Xd/SP*

Is the 64-bit name of the destination general-purpose register or stack pointer.

### Operation

Move (bitmask immediate) writes a bitmask immediate value to a register.

*Rd* = *imm*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.105 MOV (register)

Move (register).

This instruction is an alias of ORR (shifted register).

The equivalent instruction is ORR *Wd*, WZR, *Wm*.

### Syntax

MOV *Wd*, *Wm* ; 32-bit

MOV *Xd*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wm*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xm*

Is the 64-bit name of the general-purpose source register.

### Operation

Move (register) copies the value in a source register to the destination register.

*Rd* = *Rm*, where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.106 MOVK

Move wide with keep.

### Syntax

MOVK *Wd*, #*imm*{, LSL #*shift*} ; 32-bit

MOVK *Xd*, #*imm*{, LSL #*shift*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*shift*

Depends on the instruction variant:

#### 32-bit general registers

Is the amount by which to shift the immediate left, either 0 (the default) or 16.

#### 64-bit general registers

Is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*imm*

Is the 16-bit unsigned immediate, in the range 0 to 65535.

### Operation

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

*Rd*<*shift+15:shift*> = *imm16*, where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.107 MOVN

Move wide with NOT.

This instruction is used by the alias MOV (inverted wide immediate).

### Syntax

```
MOVN Wd, #imm{, LSL #shift} ; 32-bit  
MOVN Xd, #imm{, LSL #shift} ; 64-bit
```

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*shift*

Depends on the instruction variant:

#### 32-bit general registers

Is the amount by which to shift the immediate left, either 0 (the default) or 16.

#### 64-bit general registers

Is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*imm*

Is the 16-bit unsigned immediate, in the range 0 to 65535.

### Operation

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

*Rd* = NOT (LSL (*imm16*, *shift*)), where *R* is either W or X.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.108 MOVZ

Move wide with zero.

This instruction is used by the alias `MOV` (wide immediate).

### Syntax

```
MOVZ Wd, #imm{, LSL #shift} ; 32-bit
```

```
MOVZ Xd, #imm{, LSL #shift} ; 64-bit
```

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*shift*

Depends on the instruction variant:

#### 32-bit general registers

Is the amount by which to shift the immediate left, either 0 (the default) or 16.

#### 64-bit general registers

Is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*imm*

Is the 16-bit unsigned immediate, in the range 0 to 65535.

### Operation

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

*Rd* = LSL (*imm16*, *shift*), where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.109 MRS

Move System Register.

### Syntax

MRS *Xt*, (*systemreg* | *Sop0\_op1\_Cn\_Cm\_op2*)

Where:

**Xt**

Is the 64-bit name of the general-purpose destination register.

**systemreg**

Is a System register name.

**op0**

Is an unsigned immediate, and can be either 2 or 3.

**op1**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Cn**

Is a name Cn, with n in the range 0 to 15.

**Cm**

Is a name Cm, with m in the range 0 to 15.

**op2**

Is a 3-bit unsigned immediate, in the range 0 to 7.

### Usage

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.110 MSR (immediate)

Move immediate value to Special Register.

### Syntax

`MSR pstatefield, #imm`

Where:

***pstatefield***

Is a PSTATE field name, and can be one of UAO, PAN, SPSel, DAIFSet or DAIFClr.

***imm***

Is a 4-bit unsigned immediate, in the range 0 to 15.

### Usage

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see *Process state, PSTATE* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

The bits that can be written are D, A, I, F, and SP. This set of bits is expanded in extensions to the architecture as follows:

- Armv8.1 adds the PAN bit.
- Armv8.2 adds the UAO bit.

***Related reference***

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

***Related information***

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.111 MSR (register)

Move general-purpose register to System Register.

### Syntax

`MSR (systemreg|Sop0_op1_Cn_Cm_op2), Xt`

Where:

**systemreg**

Is a System register name.

**op0**

Is an unsigned immediate, and can be either 2 or 3.

**op1**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Cn**

Is a name Cn, with n in the range 0 to 15.

**Cm**

Is a name Cm, with m in the range 0 to 15.

**op2**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Xt**

Is the 64-bit name of the general-purpose source register.

### Usage

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.112 MSUB

Multiply-Subtract.

This instruction is used by the alias MNEG.

### Syntax

MSUB *Wd*, *Wn*, *Wm*, *Wa* ; 32-bit

MSUB *Xd*, *Xn*, *Xm*, *Xa* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

*Wa*

Is the 32-bit name of the third general-purpose source register holding the minuend.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

*Xm*

Is the 64-bit name of the second general-purpose source register holding the multiplier.

*Xa*

Is the 64-bit name of the third general-purpose source register holding the minuend.

### Operation

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

$Rd = Ra - Rn * Rm$ , where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.113 MUL

Multiply.

This instruction is an alias of MADD.

The equivalent instruction is MADD *Rd*, *Rn*, *Rm*, WZR.

### Syntax

MUL *Rd*, *Rn*, *Rm* ; 32-bit

MUL *Rd*, *Rn*, *Rm* ; 64-bit

Where:

***Rd***

Is the 32-bit name of the general-purpose destination register.

***Rn***

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

***Rm***

Is the 32-bit name of the second general-purpose source register holding the multiplier.

***Rd***

Is the 64-bit name of the general-purpose destination register.

***Rn***

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

***Rm***

Is the 64-bit name of the second general-purpose source register holding the multiplier.

### Operation

*Rd* = *Rn* \* *Rm*, where *R* is either W or X.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.114 MVN

Bitwise NOT.

This instruction is an alias of ORN (shifted register).

The equivalent instruction is ORN *Wd*, WZR, *Wm{*, *shift #amount}**}*.

### Syntax

MVN *Wd*, *Wm{*, *shift #amount}**}* ; 32-bit

MVN *Xd*, *Xm{*, *shift #amount}**}* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wm*

Is the 32-bit name of the general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xm*

Is the 64-bit name of the general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

*Rd* = NOT *shift(Rm, amount)*, where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.115 NEG (shifted register)

Negate (shifted register).

This instruction is an alias of SUB (shifted register).

The equivalent instruction is SUB *Rd*, WZR, *Rm* {, *shift #amount*}.

### Syntax

NEG *Rd*, *Rm*{, *shift #amount*} ; 32-bit

NEG *Xd*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Rd*

Is the 32-bit name of the general-purpose destination register.

*Rm*

Is the 32-bit name of the general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xm*

Is the 64-bit name of the general-purpose source register.

*shift*

Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

### Operation

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

*Rd* = 0 - *shift(Rm, amount)*, where *R* is either W or X.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.116 NEGS

Negate, setting flags.

This instruction is an alias of SUBS (shifted register).

The equivalent instruction is SUBS *Rd*, WZR, *Rm* {, *shift #amount*}.

### Syntax

NEGS *Rd*, *Rm*{, *shift #amount*} ; 32-bit

NEGS *Xd*, *Xm*{, *shift #amount*} ; 64-bit

Where:

***Rd***

Is the 32-bit name of the general-purpose destination register.

***Rm***

Is the 32-bit name of the general-purpose source register.

***amount***

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

***Xd***

Is the 64-bit name of the general-purpose destination register.

***Xm***

Is the 64-bit name of the general-purpose source register.

***shift***

Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

### Operation

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

$Rd = \theta - shift(Rm, amount)$ , where  $R$  is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.117 NGC

Negate with Carry.

This instruction is an alias of SBC.

The equivalent instruction is SBC *Wd*, WZR, *Wm*.

### Syntax

NGC *Wd*, *Wm* ; 32-bit

NGC *Xd*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wm*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xm*

Is the 64-bit name of the general-purpose source register.

### Operation

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

$Rd = 0 - Rm - 1 + C$ , where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.118 NGCS

Negate with Carry, setting flags.

This instruction is an alias of SBCS.

The equivalent instruction is SBCS *Wd*, WZR, *Wm*.

### Syntax

NGCS *Wd*, *Wm* ; 32-bit

NGCS *Xd*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wm*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xm*

Is the 64-bit name of the general-purpose source register.

### Operation

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

$Rd = 0 - Rm - 1 + C$ , where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.119 NOP

No Operation.

### Usage

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

---

### Note

---

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.

---

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.120 ORN (shifted register)

Bitwise OR NOT (shifted register).

This instruction is used by the alias MVN.

### Syntax

ORN *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

ORN *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

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.

*Rd* = *Rn* OR NOT *shift(Rm, amount)*, where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.121 ORR (immediate)

Bitwise OR (immediate).

This instruction is used by the alias `MOV` (bitmask immediate).

### Syntax

`ORR Wd/WSP, Wn, #imm ; 32-bit`

`ORR Xd/SP, Xn, #imm ; 64-bit`

Where:

***Wd/WSP***

Is the 32-bit name of the destination general-purpose register or stack pointer.

***Wn***

Is the 32-bit name of the general-purpose source register.

***imm***

The bitmask immediate.

***Xd/SP***

Is the 64-bit name of the destination general-purpose register or stack pointer.

***Xn***

Is the 64-bit name of the general-purpose source register.

### Operation

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.

$Rd = Rn \text{ OR } imm$ , where  $R$  is either `W` or `X`.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.122 ORR (shifted register)

Bitwise OR (shifted register).

This instruction is used by the alias MOV (register).

### Syntax

ORR *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

ORR *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

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.

*Rd* = *Rn* OR *shift(Rm, amount)*, where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.123 PACDA, PACDZA

Pointer Authentication Code for Data address, using key A.

### Syntax

PACDA  $Xd$ ,  $Xn/SP$  ; PACDA general registers

PACDZA  $Xd$  ; PACDZA general registers

Where:

$Xn/SP$

Is the 64-bit name of the general-purpose source register or stack pointer.

$Xd$

Is the 64-bit name of the general-purpose destination register.

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by  $Xd$ .

The modifier is:

- In the general-purpose register or stack pointer that is specified by  $Xn/SP$  for PACDA.
- The value zero, for PACDZA.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.124 PACDB, PACDZB

Pointer Authentication Code for Data address, using key B.

### Syntax

PACDB  $Xd$ ,  $Xn/SP$  ; PACDB general registers

PACDZB  $Xd$  ; PACDZB general registers

Where:

$Xn/SP$

Is the 64-bit name of the general-purpose source register or stack pointer.

$Xd$

Is the 64-bit name of the general-purpose destination register.

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by  $Xd$ .

The modifier is:

- In the general-purpose register or stack pointer that is specified by  $Xn/SP$  for PACDB.
- The value zero, for PACDZB.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.125 PACGA

Pointer Authentication Code, using Generic key.

### Syntax

PACGA  $Xd$ ,  $Xn$ ,  $Xm/SP$

Where:

$Xd$

Is the 64-bit name of the general-purpose destination register.

$Xn$

Is the 64-bit name of the first general-purpose source register.

$Xm/SP$

Is the 64-bit name of the second general-purpose source register or stack pointer.

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.126 PACIA, PACIZA, PACIA1716, PACIASP, PACIAZ

Pointer Authentication Code for Instruction address, using key A.

### Syntax

PACIA  $Xd$ ,  $Xn/SP$  ; PACIA general registers

PACIZA  $Xd$  ; PACIZA general registers

PACIA1716 ; PACIA1716

PACIASP ; PACIASP

PACIAZ ; PACIAZ

Where:

$Xd$

Is the 64-bit name of the general-purpose destination register.

$Xn/SP$

Is the 64-bit name of the general-purpose source register or stack pointer.

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by  $Xd$  for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by  $Xn/SP$  for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.127 PACIB, PACIZB, PACIB1716, PACIBSP, PACIBZ

Pointer Authentication Code for Instruction address, using key B.

### Syntax

PACIB  $Xd$ ,  $Xn/SP$  ; PACIB general registers

PACIZB  $Xd$  ; PACIZB general registers

PACIB1716 ; PACIB1716

PACIBSP ; PACIBSP

PACIBZ ; PACIBZ

Where:

$Xd$

Is the 64-bit name of the general-purpose destination register.

$Xn/SP$

Is the 64-bit name of the general-purpose source register or stack pointer.

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by  $Xd$  for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by  $Xn/SP$  for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.128 PSB

Profiling Synchronization Barrier.

### Syntax

PSB CSYNC

### Architectures supported

Supported in the Armv8.2 architecture and later.

### Usage

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.

### *Related reference*

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.129 RBIT

Reverse Bits.

### Syntax

RBIT *Wd*, *Wn* ; 32-bit

RBIT *Xd*, *Xn* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

Reverse Bits reverses the bit order in a register.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.130 RET

Return from subroutine.

### Syntax

RET {*Xn*}

Where:

*Xn*

Is the 64-bit name of the general-purpose register holding the address to be branched to.  
Defaults to X30 if absent.

### Usage

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

### *Related reference*

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.131 RETAA, RETAB

Return from subroutine, with pointer authentication.

### Syntax

RETAA ; RETAA general registers

RETAB ; RETAB general registers

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for RETAA, and key B is used for RETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to LR.

### *Related reference*

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.132 REV16

Reverse bytes in 16-bit halfwords.

### Syntax

REV16 *Wd*, *Wn* ; 32-bit

REV16 *Xd*, *Xn* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.133 REV32

Reverse bytes in 32-bit words.

### Syntax

REV32 *Xd*, *Xn*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.134 REV64

Reverse Bytes.

This instruction is an alias of REV.

The equivalent instruction is REV *Xd, Xn*.

### Syntax

REV64 *Xd, Xn*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.

When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.135 REV

Reverse Bytes.

This instruction is used by the alias REV64.

### Syntax

REV *Wd*, *Wn* ; 32-bit

REV *Xd*, *Xn* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

Reverse Bytes reverses the byte order in a register.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.136 ROR (immediate)

Rotate right (immediate).

This instruction is an alias of EXTR.

The equivalent instruction is EXTR *Wd*, *Ws*, *Ws*, #*shift*.

### Syntax

ROR *Wd*, *Ws*, #*shift* ; 32-bit

ROR *Xd*, *Xs*, #*shift* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Ws*

Is the 32-bit name of the general-purpose source register.

*shift*

Depends on the instruction variant:

#### 32-bit general registers

Is the amount by which to rotate, in the range 0 to 31.

#### 64-bit general registers

Is the amount by which to rotate, in the range 0 to 63.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xs*

Is the 64-bit name of the general-purpose source register.

### Operation

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

*Rd* = ROR(*Rs*, *shift*), where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.137 ROR (register)

Rotate Right (register).

This instruction is an alias of RORV.

The equivalent instruction is RORV *Wd*, *Wn*, *Wm*.

### Syntax

ROR *Wd*, *Wn*, *Wm* ; 32-bit

ROR *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

### Operation

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

*Rd* = ROR(*Rn*, *Rm*), where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.138 RORV

Rotate Right Variable.

This instruction is used by the alias ROR (register).

### Syntax

RORV *Wd*, *Wn*, *Wm* ; 32-bit

RORV *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

### Operation

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

*Rd* = ROR(*Rn*, *Rm*), where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.139 SBC

Subtract with Carry.

This instruction is used by the alias NGC.

### Syntax

SBC *Wd*, *Wn*, *Wm* ; 32-bit

SBC *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

### Operation

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.

$Rd = Rn - Rm - 1 + C$ , where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.140 SBCS

Subtract with Carry, setting flags.

This instruction is used by the alias NGCS.

### Syntax

SBCS *Wd*, *Wn*, *Wm* ; 32-bit

SBCS *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

### Operation

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.

$Rd = Rn - Rm - 1 + C$ , where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.141 SBFIZ

Signed Bitfield Insert in Zero.

This instruction is an alias of **SBFM**.

The equivalent instruction is **SBFM Wd, Wn, #(-Lsb MOD 32), #(width-1)**.

### Syntax

**SBFIZ Wd, Wn, #Lsb, #width ; 32-bit**

**SBFIZ Xd, Xn, #Lsb, #width ; 64-bit**

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Wn**

Is the 32-bit name of the general-purpose source register.

**Lsb**

Depends on the instruction variant:

#### 32-bit general registers

Is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

#### 64-bit general registers

Is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

**width**

Depends on the instruction variant:

#### 32-bit general registers

Is the width of the bitfield, in the range 1 to 32-Lsb.

#### 64-bit general registers

Is the width of the bitfield, in the range 1 to 64-Lsb.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Xn**

Is the 64-bit name of the general-purpose source register.

### Usage

Signed Bitfield Insert in Zero zeros the destination register and copies any number of contiguous bits from a source register into any position in the destination register, sign-extending the most significant bit of the transferred value.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.142 SBFM

Signed Bitfield Move.

This instruction is used by the aliases:

- ASR (immediate).
- SBFIZ.
- SBFX.
- SXTB.
- SXTH.
- SXTW.

### Syntax

SBFM *Wd*, *Wh*, #<immr>, #<imms> ; 32-bit

SBFM *Xd*, *Xn*, #<immr>, #<imms> ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wh*

Is the 32-bit name of the general-purpose source register.

<immr>

Depends on the instruction variant:

#### 32-bit general registers

Is the right rotate amount, in the range 0 to 31.

#### 64-bit general registers

Is the right rotate amount, in the range 0 to 63.

<imms>

Depends on the instruction variant:

#### 32-bit general registers

Is the leftmost bit number to be moved from the source, in the range 0 to 31.

#### 64-bit general registers

Is the leftmost bit number to be moved from the source, in the range 0 to 63.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

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.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.143 SBFX

Signed Bitfield Extract.

This instruction is an alias of **SBFM**.

The equivalent instruction is **SBFM Wd, Wn, #Lsb, #(Lsb+width-1)**.

### Syntax

**SBFX Wd, Wn, #Lsb, #width ; 32-bit**

**SBFX Xd, Xn, #Lsb, #width ; 64-bit**

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Wn**

Is the 32-bit name of the general-purpose source register.

**Lsb**

Depends on the instruction variant:

#### 32-bit general registers

Is the bit number of the lsb of the source bitfield, in the range 0 to 31.

#### 64-bit general registers

Is the bit number of the lsb of the source bitfield, in the range 0 to 63.

**width**

Depends on the instruction variant:

#### 32-bit general registers

Is the width of the bitfield, in the range 1 to 32-*Lsb*.

#### 64-bit general registers

Is the width of the bitfield, in the range 1 to 64-*Lsb*.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Xn**

Is the 64-bit name of the general-purpose source register.

### Usage

Signed Bitfield Extract extracts any number of adjacent bits at any position from a register, sign-extends them to the size of the register, and writes the result to the destination register.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.144 SDIV

Signed Divide.

### Syntax

SDIV *Wd*, *Wn*, *Wm* ; 32-bit

SDIV *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

### Operation

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.

*Rd* = *Rn* / *Rm*, where *R* is either W or X.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.145 SEV

Send Event.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D2.146 SEVL

Send Event Local.

### Usage

Send Event Local is a hint instruction. It causes an event to be signaled locally without the requirement to affect other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.147 SMADDL

Signed Multiply-Add Long.

This instruction is used by the alias SMULL.

### Syntax

SMADDL *Xd*, *Wn*, *Wm*, *Xa*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

*Xa*

Is the 64-bit name of the third general-purpose source register holding the addend.

### Operation

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.

$Xd = Xa + Wn * Wm$ .

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.148 SMC

Supervisor call to allow OS or Hypervisor code to call the Secure Monitor. It generates an exception targeting exception level 3 (EL3).

### Syntax

SMC #*imm*

Where:

*imm*

Is a 16-bit unsigned immediate, in the range 0 to 65535. This value is made available to the handler in the Exception Syndrome Register.

### Usage

Secure Monitor Call causes an exception to EL3.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of HCR\_EL2.TSC and SCR\_EL3.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR\_ELx, using the EC value 0x17, that is taken to EL3.

If the value of HCR\_EL2.TSC is 1, execution of an SMC instruction in a Non-secure EL1 state generates an exception that is taken to EL2, regardless of the value of SCR\_EL3.SMD. For more information, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

If the value of HCR\_EL2.TSC is 0 and the value of SCR\_EL3.SMD is 1, the SMC instruction is UNDEFINED.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.149 SMNEGL

Signed Multiply-Negate Long.

This instruction is an alias of SMSUBL.

The equivalent instruction is SMSUBL *Xd*, *Wn*, *Wm*, XZR.

### Syntax

SMNEGL *Xd*, *Wn*, *Wm*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

### Operation

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

*Xd* = -(*Wn* \* *Wm*).

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.150 SMSUBL

Signed Multiply-Subtract Long.

This instruction is used by the alias SMNEGL.

### Syntax

`SMSUBL Xd, Wn, Wm, Xa`

Where:

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Wn**

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

**Wm**

Is the 32-bit name of the second general-purpose source register holding the multiplier.

**Xa**

Is the 64-bit name of the third general-purpose source register holding the minuend.

### Operation

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.

$Xd = Xa - Wn * Wm$ .

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.151 SMULH

Signed Multiply High.

### Syntax

SMULH  $Xd$ ,  $Xn$ ,  $Xm$

Where:

$Xd$

Is the 64-bit name of the general-purpose destination register.

$Xn$

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

$Xm$

Is the 64-bit name of the second general-purpose source register holding the multiplier.

### Operation

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.

$Xd = \text{bits}\langle 127:64 \rangle \text{ of } Xn * Xm.$

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.152 SMULL

Signed Multiply Long.

This instruction is an alias of SMADDL.

The equivalent instruction is SMADDL *Xd*, *Wn*, *Wm*, XZR.

### Syntax

SMULL *Xd*, *Wn*, *Wm*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

### Operation

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

$Xd = Wn * Wm$ .

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.153 ST2G

Store Allocation Tags.

### Syntax

ST2G [*Xn/SP*], #*simm* ; Post-index

ST2G [*Xn/SP*, #*simm*]! ; Pre-index

ST2G [*Xn/SP{*, #*simm*}] ; Signed offset

Where:

#### *Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

#### *simm*

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.154 STG

Store Allocation Tag.

### Syntax

```
STG [Xn/SP], #simm ; Post-index  
STG [Xn/SP, #simm]! ; Pre-index  
STG [Xn/SP{, #simm}] ; Signed offset
```

Where:

#### Xn/SP

Is the 64-bit name of the general-purpose base register or stack pointer.

#### simm

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.155 STGP

Store Allocation Tag and Pair of registers.

### Syntax

```
STGP Xt1, Xt2, [Xn/SP], #imm ; Post-index  
STGP Xt1, Xt2, [Xn/SP, #imm]! ; Pre-index  
STGP Xt1, Xt2, [Xn/SP{, #imm}] ; Signed offset
```

Where:

*imm*

Depends on the instruction variant:

#### Post-index and Pre-index general registers

Is the signed immediate offset, in the range -64 to 63.

#### Pre-index general registers

Is the signed immediate offset, in the range -64 to 63.

#### Signed offset general registers

Is the optional signed immediate offset, in the range -64 to 63, defaulting to 0.

*Xt1*

Is the 64-bit name of the first general-purpose register to be transferred.

*Xt2*

Is the 64-bit name of the second general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.156 STGV

Store Tag Vector.

### Syntax

STGV *Xt*, [*Xn/SP*]!

Where:

***Xt***

Is the 64-bit name of the general-purpose register to be transferred.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Store Tag Vector reads from the second source register an IMPLEMENTATION DEFINED number of Allocation Tags and stores them to the naturally aligned array of 16 allocation tags which includes a tag whose address is the address in the first source register. The Allocation Tag at the address in the first source register is always stored, and the first source register is updated to the address of the first Allocation Tag at an address higher than the original address that was not loaded.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.157 STZ2G

Store Allocation Tags, Zeroing.

### Syntax

```
STZ2G [Xn/SP], #simm ; Post-index  
STZ2G [Xn/SP, #simm]! ; Pre-index  
STZ2G [Xn/SP{, #simm}] ; Signed offset
```

Where:

#### Xn/SP

Is the 64-bit name of the general-purpose base register or stack pointer.

#### simm

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.158 STZG

Store Allocation Tag, Zeroing.

### Syntax

STZG [*Xn/SP*], #*simm* ; Post-index

STZG [*Xn/SP*, #*simm*]! ; Pre-index

STZG [*Xn/SP{, #simm}*] ; Signed offset

Where:

#### *Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

#### *simm*

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.159 SUB (extended register)

Subtract (extended register).

### Syntax

`SUB Wd/WSP, Wn/WSP, Wm{, extend {#amount}} ; 32-bit`

`SUB Xd/SP, Xn/SP, Rm{, extend {#amount}} ; 64-bit`

Where:

**Wd/WSP**

Is the 32-bit name of the destination general-purpose register or stack pointer.

**Wn/WSP**

Is the 32-bit name of the first source general-purpose register or stack pointer.

**Wm**

Is the 32-bit name of the second general-purpose source register.

**extend**

Is the extension to be applied to the second source operand:

#### 32-bit general registers

Can be one of UXTB, UXTH, LSL|UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rd* or *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

#### 64-bit general registers

Can be one of UXTB, UXTH, UXTW, LSL|UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rd* or *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

**Xd/SP**

Is the 64-bit name of the destination general-purpose register or stack pointer.

**Xn/SP**

Is the 64-bit name of the first source general-purpose register or stack pointer.

**R**

Is a width specifier, and can be either W or X.

**m**

Is the number [0-30] of the second general-purpose source register or the name ZR (31).

**amount**

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. 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

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.

$Rd = Rn - LSL(extend(Rm), amount)$ , where *R* is either W or X.

## Usage

**Table D2-9 SUB (64-bit general registers) specifier combinations**

<i>R</i>	<i>extend</i>
W	SXTB
W	SXTH
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL   UXTX
X	SXTX

### *Related reference*

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.160 SUB (immediate)

Subtract (immediate).

### Syntax

`SUB Wd/WSP, Wn/WSP, #imm{, shift} ; 32-bit`

`SUB Xd/SP, Xn/SP, #imm{, shift} ; 64-bit`

Where:

***Wd/WSP***

Is the 32-bit name of the destination general-purpose register or stack pointer.

***Wn/WSP***

Is the 32-bit name of the source general-purpose register or stack pointer.

***Xd/SP***

Is the 64-bit name of the destination general-purpose register or stack pointer.

***Xn/SP***

Is the 64-bit name of the source general-purpose register or stack pointer.

***imm***

Is an unsigned immediate, in the range 0 to 4095.

***shift***

Is the optional left shift to apply to the immediate, defaulting to LSL #0, and can be either LSL #0 or LSL #12.

### Operation

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

$Rd = Rn - \text{shift}(\text{imm})$ , where  $R$  is either  $W$  or  $X$ .

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.161 SUB (shifted register)

Subtract (shifted register).

This instruction is used by the alias NEG (shifted register).

### Syntax

`SUB Wd, Wn, Wm{, shift #amount} ; 32-bit`

`SUB Xd, Xn, Xm{, shift #amount} ; 64-bit`

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Wn**

Is the 32-bit name of the first general-purpose source register.

**Wm**

Is the 32-bit name of the second general-purpose source register.

**amount**

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Xn**

Is the 64-bit name of the first general-purpose source register.

**Xm**

Is the 64-bit name of the second general-purpose source register.

**shift**

Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

### Operation

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

$Rd = Rn - \text{shift}(Rm, amount)$ , where  $R$  is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.162 SUBG

Subtract with Tag.

### Syntax

SUBG *Xd/SP*, *Xn/SP*, #<uimm6>, #<uimm4>

Where:

***Xd/SP***

Is the 64-bit name of the destination general-purpose register or stack pointer.

***Xn/SP***

Is the 64-bit name of the source general-purpose register or stack pointer.

**<uimm6>**

Is an unsigned immediate, a multiple of 16 in the range 0 to 1008.

**<uimm4>**

Is an unsigned immediate, in the range 0 to 15.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR\_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.163 SUBP

Subtract Pointer.

### Syntax

SUBP  $Xd$ ,  $Xn/SP$ ,  $Xm/SP$

Where:

$Xd$

Is the 64-bit name of the general-purpose destination register.

$Xn/SP$

Is the 64-bit name of the first source general-purpose register or stack pointer.

$Xm/SP$

Is the 64-bit name of the second general-purpose source register or stack pointer.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.164 SUBPS

Subtract Pointer, setting Flags.

This instruction is used by the alias CMPP.

### Syntax

SUBPS  $Xd$ ,  $Xn/SP$ ,  $Xm/SP$

Where:

$Xd$

Is the 64-bit name of the general-purpose destination register.

$Xn/SP$

Is the 64-bit name of the first source general-purpose register or stack pointer.

$Xm/SP$

Is the 64-bit name of the second general-purpose source register or stack pointer.

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

### Related reference

[D2.59 CMPP on page D2-731](#)

[D2.1 A64 instructions in alphabetical order on page D2-662](#)

## D2.165 SUBS (extended register)

Subtract (extended register), setting flags.

This instruction is used by the alias **CMP** (extended register).

### Syntax

`SUBS Wd, Wn/WSP, Wm{, extend {#amount}} ; 32-bit`

`SUBS Xd, Xn/SP, Rm{, extend {#amount}} ; 64-bit`

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Wn/WSP**

Is the 32-bit name of the first source general-purpose register or stack pointer.

**Wm**

Is the 32-bit name of the second general-purpose source register.

**extend**

Is the extension to be applied to the second source operand:

#### 32-bit general registers

Can be one of UXTB, UXTH, LSL | UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

#### 64-bit general registers

Can be one of UXTB, UXTH, UXTW, LSL | UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Xn/SP**

Is the 64-bit name of the first source general-purpose register or stack pointer.

**R**

Is a width specifier, and can be either W or X.

**m**

Is the number [0-30] of the second general-purpose source register or the name ZR (31).

**amount**

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. 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

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.

$Rd = Rn - LSL(\text{extend}(Rm), \text{amount})$ , where *R* is either W or X.

## Usage

Table D2-10 SUBS (64-bit general registers) specifier combinations

<i>R</i>	<i>extend</i>
W	SXTB
W	SXTH
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL   UXTX
X	SXTX

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.166 SUBS (immediate)

Subtract (immediate), setting flags.

This instruction is used by the alias **CMP** (immediate).

### Syntax

`SUBS Wd, Wn/WSP, #imm{, shift} ; 32-bit`

`SUBS Xd, Xn/SP, #imm{, shift} ; 64-bit`

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Wn/WSP**

Is the 32-bit name of the source general-purpose register or stack pointer.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Xn/SP**

Is the 64-bit name of the source general-purpose register or stack pointer.

**imm**

Is an unsigned immediate, in the range 0 to 4095.

**shift**

Is the optional left shift to apply to the immediate, defaulting to `LSL #0`, and can be either `LSL #0` or `LSL #12`.

### Operation

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.

$Rd = Rn - \text{shift}(\text{imm})$ , where  $R$  is either  $W$  or  $X$ .

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.167 SUBS (shifted register)

Subtract (shifted register), setting flags.

This instruction is used by the aliases:

- CMP (shifted register).
- NEGS.

### Syntax

SUBS *Wd*, *Wn*, *Wm*{, *shift #amount*} ; 32-bit

SUBS *Xd*, *Xn*, *Xm*{, *shift #amount*} ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

### Operation

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.

$Rd = Rn - shift(Rm, amount)$ , where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.168 SVC

Supervisor call to allow application code to call the OS. It generates an exception targeting exception level 1 (EL1).

### Syntax

SVC #imm

Where:

imm

Is a 16-bit unsigned immediate, in the range 0 to 65535. This value is made available to the handler in the Exception Syndrome Register.

### Usage

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in ESR\_ELx, using the EC value 0x15, and the value of the immediate argument.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.169 SXTB

Signed Extend Byte.

This instruction is an alias of **SBFM**.

The equivalent instruction is **SBFM Wd, Wn, #0, #7**.

### Syntax

**SXTB Wd, Wn ; 32-bit**

**SXTB Xd, Wn ; 64-bit**

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Wn**

Is the 32-bit name of the general-purpose source register.

### Operation

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

$Rd = \text{SignExtend}(Wn<7:0>)$ , where  $R$  is either **W** or **X**.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.170 SXT<sub>H</sub>

Sign Extend Halfword.

This instruction is an alias of SBFM.

The equivalent instruction is SBFM *Wd*, *Wn*, #0, #15.

### Syntax

SXT<sub>H</sub> *Wd*, *Wn* ; 32-bit

SXT<sub>H</sub> *Xd*, *Wn* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

### Operation

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

*Rd* = SignExtend(*Wn*<15:0>), where *R* is either W or X.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.171 SXTW

Sign Extend Word.

This instruction is an alias of **SBFM**.

The equivalent instruction is **SBFM** *Xd*, *Xn*, #0, #31.

### Syntax

**SXTW** *Xd*, *Wn*

Where:

***Xd***

Is the 64-bit name of the general-purpose destination register.

***Xn***

Is the 64-bit name of the general-purpose source register.

***Wn***

Is the 32-bit name of the general-purpose source register.

### Operation

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

*Xd* = *SignExtend(Wn<31:0>)*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.172 SYS

System instruction.

This instruction is used by the aliases:

- AT.
- DC.
- IC.
- TLBI.

### Syntax

`SYS #op1, Cn, Cm, #op2{, Xt}`

Where:

*op1*

Is a 3-bit unsigned immediate, in the range 0 to 7.

*Cn*

Is a name *Cn*, with *n* in the range 0 to 15.

*Cm*

Is a name *Cm*, with *m* in the range 0 to 15.

*op2*

Is a 3-bit unsigned immediate, in the range 0 to 7.

*Xt*

Is the 64-bit name of the optional general-purpose source register, defaulting to 31.

### Usage

System instruction. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* in the [Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile](#) for the encodings of System instructions.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.173 SYSL

System instruction with result.

### Syntax

**SYSL** *Xt*, #*op1*, *Cn*, *Cm*, #*op2*

Where:

**Xt**

Is the 64-bit name of the general-purpose destination register.

**op1**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Cn**

Is a name *Cn*, with *n* in the range 0 to 15.

**Cm**

Is a name *Cm*, with *m* in the range 0 to 15.

**op2**

Is a 3-bit unsigned immediate, in the range 0 to 7.

### Usage

System instruction with result. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#) for the encodings of System instructions.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.174 TBNZ

Test bit and Branch if Nonzero.

### Syntax

TBNZ *R<t>*, #*imm*, *Label*

Where:

*R*

Is a width specifier, and can be either W or 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).

*imm*

Is the bit number to be tested, in the range 0 to 63.

*Label*

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range ±32KB.

### Usage

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.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.175 TBZ

Test bit and Branch if Zero.

### Syntax

`TBZ R<t>, #imm, Label`

Where:

**R**

Is a width specifier, and can be either W or 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).

**imm**

Is the bit number to be tested, in the range 0 to 63.

**Label**

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range  $\pm 32\text{KB}$ .

### Usage

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.

### Related reference

[D2.1 A64 instructions in alphabetical order on page D2-662](#)

## D2.176 TLBI

TLB Invalidate operation.

This instruction is an alias of **SYS**.

The equivalent instruction is **SYS #op1, C8, Cm, #op2{, Xt}**.

### Syntax

**TLBI <tlbi\_op>{, Xt}**

Where:

**op1**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**Cm**

Is a name **Cm**, with **m** in the range 0 to 15.

**op2**

Is a 3-bit unsigned immediate, in the range 0 to 7.

**<tlbi\_op>**

Is a TLBI instruction name, as listed for the TLBI system instruction group, and can be one of the values shown in Usage.

**Xt**

Is the 64-bit name of the optional general-purpose source register, defaulting to 31.

### Usage

TLB Invalidate operation. For more information, see *A64 system instructions for TLB maintenance* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The following table shows the valid specifier combinations:

**Table D2-11 SYS parameter values corresponding to TLBI operations**

<b>&lt;tlbi_op&gt;</b>	<b>op1</b>	<b>Cm</b>	<b>op2</b>
ALLE1	4	7	4
ALLE1IS	4	3	4
ALLE2	4	7	0
ALLE2IS	4	3	0
ALLE3	6	7	0
ALLE3IS	6	3	0
ASIDE1	0	7	2
ASIDE1IS	0	3	2
IPAS2E1	4	4	1
IPAS2E1IS	4	0	1
IPAS2LE1	4	4	5
IPAS2LE1IS	4	0	5
VAAE1	0	7	3
VAAE1IS	0	3	3
VAALE1	0	7	7

Table D2-11 SYS parameter values corresponding to TLBI operations (continued)

<tlbi_op>	op1	Cm	op2
VAALE1IS	0	3	7
VAE1	0	7	1
VAE1IS	0	3	1
VAE2	4	7	1
VAE2IS	4	3	1
VAE3	6	7	1
VAE3IS	6	3	1
VALE1	0	7	5
VALE1IS	0	3	5
VALE2	4	7	5
VALE2IS	4	3	5
VALE3	6	7	5
VALE3IS	6	3	5
VMALLE1	0	7	0
VMALLE1IS	0	3	0
VMALLS12E1	4	7	6
VMALLS12E1IS	4	3	6

**Related reference**

*D2.1 A64 instructions in alphabetical order* on page D2-662

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D2.177 TST (immediate)

, setting the condition flags and discarding the result.

This instruction is an alias of ANDS (immediate).

The equivalent instruction is ANDS WZR, *Wn*, #*imm*.

### Syntax

TST *Wn*, #*imm* ; 32-bit

TST *Xn*, #*imm* ; 64-bit

Where:

*Wn*

Is the 32-bit name of the general-purpose source register.

*imm*

The bitmask immediate.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

*Rn* AND *imm*, where *R* is either *W* or *X*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.178 TST (shifted register)

Test (shifted register).

This instruction is an alias of ANDS (shifted register).

The equivalent instruction is ANDS WZR, *Rn*, *Rm{*, *shift #amount}*}.

### Syntax

TST *Rn*, *Rm{*, *shift #amount}*} ; 32-bit

TST *Xn*, *Xm{*, *shift #amount}*} ; 64-bit

Where:

*Rn*

Is the 32-bit name of the first general-purpose source register.

*Rm*

Is the 32-bit name of the second general-purpose source register.

*amount*

Depends on the instruction variant:

#### 32-bit general registers

Is the shift amount, in the range 0 to 31, defaulting to 0.

#### 64-bit general registers

Is the shift amount, in the range 0 to 63, defaulting to 0.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

*shift*

Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

### Operation

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

*Rn* AND *shift(Rm, amount)*, where *R* is either *W* or *X*.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.179 UBFIZ

Unsigned Bitfield Insert in Zero.

This instruction is an alias of UBFM.

The equivalent instruction is UBFM *Wd*, *Wn*, #(−*Lsb* MOD 32), #(width−1).

### Syntax

UBFIZ *Wd*, *Wn*, #*Lsb*, #*width* ; 32-bit

UBFIZ *Xd*, *Xn*, #*Lsb*, #*width* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Lsb*

Depends on the instruction variant:

#### 32-bit general registers

Is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

#### 64-bit general registers

Is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

*width*

Depends on the instruction variant:

#### 32-bit general registers

Is the width of the bitfield, in the range 1 to 32−*Lsb*.

#### 64-bit general registers

Is the width of the bitfield, in the range 1 to 64−*Lsb*.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

Unsigned Bitfield Insert in Zero zeros the destination register and copies any number of contiguous bits from a source register into any position in the destination register.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.180 UBFM

Unsigned Bitfield Move.

This instruction is used by the aliases:

- LSL (immediate).
- LSR (immediate).
- UBFIZ.
- UBFX.
- UXTB.
- UXTH.

### Syntax

UBFM *Wd*, *Wh*, #<immr>, #<imms> ; 32-bit

UBFM *Xd*, *Xn*, #<immr>, #<imms> ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wh*

Is the 32-bit name of the general-purpose source register.

<immr>

Depends on the instruction variant:

#### 32-bit general registers

Is the right rotate amount, in the range 0 to 31.

#### 64-bit general registers

Is the right rotate amount, in the range 0 to 63.

<imms>

Depends on the instruction variant:

#### 32-bit general registers

Is the leftmost bit number to be moved from the source, in the range 0 to 31.

#### 64-bit general registers

Is the leftmost bit number to be moved from the source, in the range 0 to 63.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

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.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.181 UBFX

Unsigned Bitfield Extract.

This instruction is an alias of UBFM.

The equivalent instruction is UBFM *Wd*, *Wn*, #*Lsb*, #(Lsb+width-1).

### Syntax

UBFX *Wd*, *Wn*, #*Lsb*, #*width* ; 32-bit

UBFX *Xd*, *Xn*, #*Lsb*, #*width* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Lsb*

Depends on the instruction variant:

#### 32-bit general registers

Is the bit number of the lsb of the source bitfield, in the range 0 to 31.

#### 64-bit general registers

Is the bit number of the lsb of the source bitfield, in the range 0 to 63.

*width*

Depends on the instruction variant:

#### 32-bit general registers

Is the width of the bitfield, in the range 1 to 32-*Lsb*.

#### 64-bit general registers

Is the width of the bitfield, in the range 1 to 64-*Lsb*.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Usage

Unsigned Bitfield Extract extracts any number of adjacent bits at any position from a register, zero-extends them to the size of the register, and writes the result to the destination register.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.182 UDIV

Unsigned Divide.

### Syntax

UDIV *Wd*, *Wn*, *Wm* ; 32-bit

UDIV *Xd*, *Xn*, *Xm* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register.

*Wm*

Is the 32-bit name of the second general-purpose source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Xn*

Is the 64-bit name of the first general-purpose source register.

*Xm*

Is the 64-bit name of the second general-purpose source register.

### Operation

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.

*Rd* = *Rn* / *Rm*, where *R* is either W or X.

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.183 UMADDL

Unsigned Multiply-Add Long.

This instruction is used by the alias UMULL.

### Syntax

UMADDL *Xd*, *Wn*, *Wm*, *Xa*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

*Xa*

Is the 64-bit name of the third general-purpose source register holding the addend.

### Operation

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.

$$Xd = Xa + Wn * Wm.$$

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.184 UMNEGL

Unsigned Multiply-Negate Long.

This instruction is an alias of UMSUBL.

The equivalent instruction is UMSUBL *Xd*, *Wn*, *Wm*, XZR.

### Syntax

UMNEGL *Xd*, *Wn*, *Wm*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

### Operation

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

*Xd* = -(*Wn* \* *Wm*).

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.185 UMSUBL

Unsigned Multiply-Subtract Long.

This instruction is used by the alias UMNEGL.

### Syntax

UMSUBL *Xd*, *Wn*, *Wm*, *Xa*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

*Xa*

Is the 64-bit name of the third general-purpose source register holding the minuend.

### Operation

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.

$$Xd = Xa - Wn * Wm.$$

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.186 UMULH

Unsigned Multiply High.

### Syntax

UMULH  $Xd$ ,  $Xn$ ,  $Xm$

Where:

$Xd$

Is the 64-bit name of the general-purpose destination register.

$Xn$

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

$Xm$

Is the 64-bit name of the second general-purpose source register holding the multiplier.

### Operation

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.

$Xd = \text{bits}\langle 127:64 \rangle \text{ of } Xn * Xm.$

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

## D2.187 UMULL

Unsigned Multiply Long.

This instruction is an alias of UMADDL.

The equivalent instruction is UMADDL *Xd*, *Wn*, *Wm*, XZR.

### Syntax

UMULL *Xd*, *Wn*, *Wm*

Where:

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

*Wm*

Is the 32-bit name of the second general-purpose source register holding the multiplier.

### Operation

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

*Xd* = *Wn* \* *Wm*.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.188 UXTB

Unsigned Extend Byte.

This instruction is an alias of UBFM.

The equivalent instruction is UBFM *Wd*, *Wn*, #0, #7.

### Syntax

UXTB *Wd*, *Wn*

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

### Operation

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

*Wd* = ZeroExtend(*Wn*<7:0>).

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.189 UXTH

Unsigned Extend Halfword.

This instruction is an alias of **UBFM**.

The equivalent instruction is **UBFM Wd, Wn, #0, #15**.

### Syntax

**UXTH Wd, Wn**

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Wn**

Is the 32-bit name of the general-purpose source register.

### Operation

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

**Wd = ZeroExtend(Wn<15:0>).**

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.190 XAFlag

Convert floating-point condition flags from external format to Arm format.

### Syntax

XAFlag

### Architectures supported

Supported in Armv8.5 and later.

### Usage

Convert floating-point condition flags from external format to Arm format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

### *Related reference*

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.191 WFE

Wait For Event.

### Usage

Wait For Event is a hint instruction that permits the PE to enter a low-power state until one of a number of events occurs, including events signaled by executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

As described in *Wait For Event mechanism and Send event* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, 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.*

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D2.192 WFI

Wait For Interrupt.

### Usage

Wait For Interrupt is a hint instruction that permits the PE to enter a low-power state until one of a number of asynchronous event occurs. For more information, see *Wait For Interrupt* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

As described in *Wait For Interrupt* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, 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.*

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D2.193 XPACD, XPACI, XPAACLRI

Strip Pointer Authentication Code.

### Syntax

XPACD *Xd* ; XPACD general registers

XPACI *Xd* ; XPACI general registers

XPAACLRI ; System

Where:

***Xd***

Is the 64-bit name of the general-purpose destination register.

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPACI and XPACD, and is in LR for XPAACLRI.

The XPACD instruction is used for data addresses, and XPACI and XPAACLRI are used for instruction addresses.

### Related reference

*D2.1 A64 instructions in alphabetical order* on page D2-662

## D2.194 YIELD

YIELD.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D2.1 A64 instructions in alphabetical order](#) on page D2-662

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

# Chapter D3

## A64 Data Transfer Instructions

Describes the A64 data transfer instructions.

It contains the following sections:

- [\*D3.1 A64 data transfer instructions in alphabetical order\*](#) on page D3-877.
- [\*D3.2 CASA, CASAL, CAS, CASL, CASAL, CAS, CASL\*](#) on page D3-883.
- [\*D3.3 CASAB, CASALB, CASB, CASLB\*](#) on page D3-884.
- [\*D3.4 CASAH, CASALH, CASH, CASLH\*](#) on page D3-885.
- [\*D3.5 CASPA, CASPAL, CASP, CASPL, CASPAL, CASP, CASPL\*](#) on page D3-886.
- [\*D3.6 LDADDA, LDADDAL, LDADD, LDADDL, LDADDAL, LDADD, LDADDL\*](#) on page D3-888.
- [\*D3.7 LDADDAB, LDADDALB, LDADDDB, LDADDLB\*](#) on page D3-889.
- [\*D3.8 LDADDAH, LDADDALH, LDADDH, LDADDLH\*](#) on page D3-890.
- [\*D3.9 LDAPR\*](#) on page D3-891.
- [\*D3.10 LDAPRB\*](#) on page D3-892.
- [\*D3.11 LDAPRH\*](#) on page D3-893.
- [\*D3.12 LDAR\*](#) on page D3-894.
- [\*D3.13 LDARB\*](#) on page D3-895.
- [\*D3.14 LDARH\*](#) on page D3-896.
- [\*D3.15 LDAXP\*](#) on page D3-897.
- [\*D3.16 LDAXR\*](#) on page D3-898.
- [\*D3.17 LDAXRB\*](#) on page D3-899.
- [\*D3.18 LDAXRH\*](#) on page D3-900.
- [\*D3.19 LDCLRA, LDCLRAL, LDCLR, LDCLRL, LDCLRAL, LDCLR, LDCLRL\*](#) on page D3-901.
- [\*D3.20 LDCLRAB, LDCLRALB, LDCLRB, LDCLRLB\*](#) on page D3-902.
- [\*D3.21 LDCLRAH, LDCLRALH, LDCLRH, LDCLRLH\*](#) on page D3-903.
- [\*D3.22 LDEORA, LDEORAL, LDEOR, LDEORL, LDEORAL, LDEOR, LDEORL\*](#) on page D3-904.
- [\*D3.23 LDEORAB, LDEORALB, LDEORB, LDEORLB\*](#) on page D3-905.

- [D3.24 LDEORAH, LDEORALH, LDEORH, LDEORLH](#) on page D3-906.
- [D3.25 LDLAR](#) on page D3-907.
- [D3.26 LDLARB](#) on page D3-908.
- [D3.27 LDLARH](#) on page D3-909.
- [D3.28 LDNP](#) on page D3-910.
- [D3.29 LDP](#) on page D3-911.
- [D3.30 LDPSW](#) on page D3-912.
- [D3.31 LDR \(immediate\)](#) on page D3-913.
- [D3.32 LDR \(literal\)](#) on page D3-914.
- [D3.33 LDR \(register\)](#) on page D3-915.
- [D3.34 LDRAA, LDRAB, LDRAB](#) on page D3-916.
- [D3.35 LDRB \(immediate\)](#) on page D3-917.
- [D3.36 LDRB \(register\)](#) on page D3-918.
- [D3.37 LDRH \(immediate\)](#) on page D3-919.
- [D3.38 LDRH \(register\)](#) on page D3-920.
- [D3.39 LDRSB \(immediate\)](#) on page D3-921.
- [D3.40 LDRSB \(register\)](#) on page D3-922.
- [D3.41 LDRSH \(immediate\)](#) on page D3-923.
- [D3.42 LDRSH \(register\)](#) on page D3-924.
- [D3.43 LDRSW \(immediate\)](#) on page D3-925.
- [D3.44 LDRSW \(literal\)](#) on page D3-926.
- [D3.45 LDRSW \(register\)](#) on page D3-927.
- [D3.46 LDSETA, LDSETAL, LDSET, LDSETL, LDSETAL, LDSET, LDSETL](#) on page D3-928.
- [D3.47 LDSETAB, LDSETALB, LDSETB, LDSETLB](#) on page D3-929.
- [D3.48 LDSETAH, LDSETALH, LDSETH, LDSETLH](#) on page D3-930.
- [D3.49 LDSMAXA, LDSMAXAL, LDSMAX, LDSMAXL, LDSMAXAL, LDSMAX, LDSMAXL](#) on page D3-931.
- [D3.50 LDSMAXAB, LDSMAXALB, LDSMAXB, LDSMAXLB](#) on page D3-932.
- [D3.51 LDSMAXAH, LDSMAXALH, LDSMAXH, LDSMAXLH](#) on page D3-933.
- [D3.52 LDSMINA, LDSMINAL, LDSMIN, LDSMINL, LDSMINAL, LDSMIN, LDSMINL](#) on page D3-934.
- [D3.53 LDSMINAB, LDSMINALB, LDSMINB, LDSMINLB](#) on page D3-935.
- [D3.54 LDSMINAH, LDSMINALH, LDSMINH, LDSMINLH](#) on page D3-936.
- [D3.55 LDTR](#) on page D3-937.
- [D3.56 LDTRB](#) on page D3-938.
- [D3.57 LDTRH](#) on page D3-939.
- [D3.58 LDTRSB](#) on page D3-940.
- [D3.59 LDTRSH](#) on page D3-941.
- [D3.60 LDTRSW](#) on page D3-942.
- [D3.61 LDUMAXA, LDUMAXAL, LDUMAX, LDUMAXL, LDUMAXAL, LDUMAX, LDUMAXL](#) on page D3-943.
- [D3.62 LDUMAXAB, LDUMAXALB, LDUMAXB, LDUMAXLB](#) on page D3-944.
- [D3.63 LDUMAXAH, LDUMAXALH, LDUMAXH, LDUMAXLH](#) on page D3-945.
- [D3.64 LDUMINA, LDUMINAL, LDUMIN, LDUMINL, LDUMINAL, LDUMIN, LDUMINL](#) on page D3-946.
- [D3.65 LDUMINAB, LDUMINALB, LDUMINB, LDUMINLB](#) on page D3-947.
- [D3.66 LDUMINAH, LDUMINALH, LDUMINH, LDUMINLH](#) on page D3-948.
- [D3.67 LDUR](#) on page D3-949.
- [D3.68 LDURB](#) on page D3-950.
- [D3.69 LDURH](#) on page D3-951.
- [D3.70 LDURSB](#) on page D3-952.
- [D3.71 LDURSH](#) on page D3-953.
- [D3.72 LDURSW](#) on page D3-954.
- [D3.73 LDXP](#) on page D3-955.
- [D3.74 LDXR](#) on page D3-956.
- [D3.75 LDXRB](#) on page D3-957.

- *D3.76 LDXRH* on page D3-958.
- *D3.77 PRFM (immediate)* on page D3-959.
- *D3.78 PRFM (literal)* on page D3-961.
- *D3.79 PRFM (register)* on page D3-963.
- *D3.80 PRFUM (unscaled offset)* on page D3-965.
- *D3.81 STADD, STADDL, STADDL* on page D3-967.
- *D3.82 STADDB, STADDL<sub>B</sub>* on page D3-968.
- *D3.83 STADDH, STADDL<sub>H</sub>* on page D3-969.
- *D3.84 STCLR, STCLRL, STCLRL* on page D3-970.
- *D3.85 STCLRB, STCLRLB* on page D3-971.
- *D3.86 STCLRH, STCLRLH* on page D3-972.
- *D3.87 STEOR, STEORL, STEORL* on page D3-973.
- *D3.88 STEORB, STEORLB* on page D3-974.
- *D3.89 STEORH, STEORLH* on page D3-975.
- *D3.90 STLLR* on page D3-976.
- *D3.91 STLLRB* on page D3-977.
- *D3.92 STLLRH* on page D3-978.
- *D3.93 STLR* on page D3-979.
- *D3.94 STLRB* on page D3-980.
- *D3.95 STLRH* on page D3-981.
- *D3.96 STLXP* on page D3-982.
- *D3.97 STLXR* on page D3-984.
- *D3.98 STLXRB* on page D3-986.
- *D3.99 STLXRH* on page D3-987.
- *D3.100 STNP* on page D3-988.
- *D3.101 STP* on page D3-989.
- *D3.102 STR (immediate)* on page D3-990.
- *D3.103 STR (register)* on page D3-991.
- *D3.104 STRB (immediate)* on page D3-992.
- *D3.105 STRB (register)* on page D3-993.
- *D3.106 STRH (immediate)* on page D3-994.
- *D3.107 STRH (register)* on page D3-995.
- *D3.108 STSET, STSETL, STSETL* on page D3-996.
- *D3.109 STSETB, STSETLB* on page D3-997.
- *D3.110 STSETH, STSETLH* on page D3-998.
- *D3.111 STSMAX, STSMAXL, STSMAXL* on page D3-999.
- *D3.112 STSMAXB, STSMAXLB* on page D3-1000.
- *D3.113 STSMAXH, STSMAXLH* on page D3-1001.
- *D3.114 STSMIN, STSMINL, STSMINL* on page D3-1002.
- *D3.115 STSMINB, STSMINLB* on page D3-1003.
- *D3.116 STSMINH, STSMINLH* on page D3-1004.
- *D3.117 STTR* on page D3-1005.
- *D3.118 STTRB* on page D3-1006.
- *D3.119 STTRH* on page D3-1007.
- *D3.120 STUMAX, STUMAXL, STUMAXL* on page D3-1008.
- *D3.121 STUMAXB, STUMAXLB* on page D3-1009.
- *D3.122 STUMAXH, STUMAXLH* on page D3-1010.
- *D3.123 STUMIN, STUMINL, STUMINL* on page D3-1011.
- *D3.124 STUMINB, STUMINLB* on page D3-1012.
- *D3.125 STUMINH, STUMINLH* on page D3-1013.
- *D3.126 STUR* on page D3-1014.
- *D3.127 STURB* on page D3-1015.
- *D3.128 STURH* on page D3-1016.
- *D3.129 STXP* on page D3-1017.
- *D3.130 STXR* on page D3-1019.
- *D3.131 STXRB* on page D3-1021.

- *D3.132 STXRH* on page D3-1022.
- *D3.133 SWPA, SWPAL, SWP, SWPL, SWPAL, SWP, SWPL* on page D3-1023.
- *D3.134 SWPAB, SWPALB, SWPB, SWPLB* on page D3-1024.
- *D3.135 SWPAH, SWPALH, SWPH, SWPLH* on page D3-1025.

## D3.1 A64 data transfer instructions in alphabetical order

A summary of the A64 data transfer instructions and pseudo-instructions that are supported.

**Table D3-1 Summary of A64 data transfer instructions**

Mnemonic	Brief description	See
CASA, CASAL, CAS, CASL, CASAL, CAS, CASL	Compare and Swap word or doubleword in memory	<a href="#">D3.2 CASA, CASAL, CAS, CASL, CASAL, CAS, CASL on page D3-883</a>
CASAB, CASALB, CASB, CASLB	Compare and Swap byte in memory	<a href="#">D3.3 CASAB, CASALB, CASB, CASLB on page D3-884</a>
CASAH, CASALH, CASH, CASLH	Compare and Swap halfword in memory	<a href="#">D3.4 CASAH, CASALH, CASH, CASLH on page D3-885</a>
CASPA, CASPAL, CASP, CASPL, CASPAL, CASP, CASPL	Compare and Swap Pair of words or doublewords in memory	<a href="#">D3.5 CASPA, CASPAL, CASP, CASPL, CASPAL, CASP, CASPL on page D3-886</a>
LDADDA, LDADDAL, LDADD, LDADDL, LDADDAL, LDADD, LDADDL	Atomic add on word or doubleword in memory	<a href="#">D3.6 LDADDA, LDADDAL, LDADD, LDADDL, LDADDAL, LDADD, LDADDL on page D3-888</a>
LDADDAB, LDADDALB, LDADDB, LDADDLB	Atomic add on byte in memory	<a href="#">D3.7 LDADDAB, LDADDALB, LDADDB, LDADDLB on page D3-889</a>
LDADDAH, LDADDALH, LDADDH, LDADDLH	Atomic add on halfword in memory	<a href="#">D3.8 LDADDAH, LDADDALH, LDADDH, LDADDLH on page D3-890</a>
LDAPR	Load-Acquire RCpc Register	<a href="#">D3.9 LDAPR on page D3-891</a>
LDAPRB	Load-Acquire RCpc Register Byte	<a href="#">D3.10 LDAPRB on page D3-892</a>
LDAPRH	Load-Acquire RCpc Register Halfword	<a href="#">D3.11 LDAPRH on page D3-893</a>
LDAR	Load-Acquire Register	<a href="#">D3.12 LDAR on page D3-894</a>
LDARB	Load-Acquire Register Byte	<a href="#">D3.13 LDARB on page D3-895</a>
LDARH	Load-Acquire Register Halfword	<a href="#">D3.14 LDARH on page D3-896</a>
LDAXP	Load-Acquire Exclusive Pair of Registers	<a href="#">D3.15 LDAXP on page D3-897</a>
LDAXR	Load-Acquire Exclusive Register	<a href="#">D3.16 LDAXR on page D3-898</a>
LDAXB	Load-Acquire Exclusive Register Byte	<a href="#">D3.17 LDAXB on page D3-899</a>
LDAXRH	Load-Acquire Exclusive Register Halfword	<a href="#">D3.18 LDAXRH on page D3-900</a>
LDCLRA, LDCLRAL, LDCLR, LDCLRL, LDCLRAL, LDCLR, LDCLRL	Atomic bit clear on word or doubleword in memory	<a href="#">D3.19 LDCLRA, LDCLRAL, LDCLR, LDCLRL, LDCLRAL, LDCLR, LDCLRL on page D3-901</a>
LDCLRAB, LDCLRALB, LDCLRB, LDCLRLB	Atomic bit clear on byte in memory	<a href="#">D3.20 LDCLRAB, LDCLRALB, LDCLRB, LDCLRLB on page D3-902</a>
LDCLRAH, LDCLRALH, LDCLRH, LDCLRLH	Atomic bit clear on halfword in memory	<a href="#">D3.21 LDCLRAH, LDCLRALH, LDCLRH, LDCLRLH on page D3-903</a>
LDEORA, LDEORAL, LDEOR, LDEORL, LDEORAL, LDEOR, LDEORL	Atomic exclusive OR on word or doubleword in memory	<a href="#">D3.22 LDEORA, LDEORAL, LDEOR, LDEORL, LDEORAL, LDEOR, LDEORL on page D3-904</a>

**Table D3-1 Summary of A64 data transfer instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
LDEORAB, LDEORALB, LDEORB, LDEORLB	Atomic exclusive OR on byte in memory	<a href="#">D3.23 LDEORAB, LDEORALB, LDEORB, LDEORLB on page D3-905</a>
LDEORAH, LDEORALH, LDEORH, LDEORLH	Atomic exclusive OR on halfword in memory	<a href="#">D3.24 LDEORAH, LDEORALH, LDEORH, LDEORLH on page D3-906</a>
LDLAR	Load LOAcquire Register	<a href="#">D3.25 LDLAR on page D3-907</a>
LDLARB	Load LOAcquire Register Byte	<a href="#">D3.26 LDLARB on page D3-908</a>
LDLARH	Load LOAcquire Register Halfword	<a href="#">D3.27 LDLARH on page D3-909</a>
LDNP	Load Pair of Registers, with non-temporal hint	<a href="#">D3.28 LDNP on page D3-910</a>
LDP	Load Pair of Registers	<a href="#">D3.29 LDP on page D3-911</a>
LDPSW	Load Pair of Registers Signed Word	<a href="#">D3.30 LDPSW on page D3-912</a>
LDR (immediate)	Load Register (immediate)	<a href="#">D3.31 LDR (immediate) on page D3-913</a>
LDR (literal)	Load Register (literal)	<a href="#">D3.32 LDR (literal) on page D3-914</a>
LDR pseudo-instruction	Load a register with either a 32-bit or 64-bit immediate value or any address	
LDR (register)	Load Register (register)	<a href="#">D3.33 LDR (register) on page D3-915</a>
LDRAA, LDRAB, LDRAB	Load Register, with pointer authentication	<a href="#">D3.34 LDRAA, LDRAB, LDRAB on page D3-916</a>
LDRB (immediate)	Load Register Byte (immediate)	<a href="#">D3.35 LDRB (immediate) on page D3-917</a>
LDRB (register)	Load Register Byte (register)	<a href="#">D3.36 LDRB (register) on page D3-918</a>
LDRH (immediate)	Load Register Halfword (immediate)	<a href="#">D3.37 LDRH (immediate) on page D3-919</a>
LDRH (register)	Load Register Halfword (register)	<a href="#">D3.38 LDRH (register) on page D3-920</a>
LDRSB (immediate)	Load Register Signed Byte (immediate)	<a href="#">D3.39 LDRSB (immediate) on page D3-921</a>
LDRSB (register)	Load Register Signed Byte (register)	<a href="#">D3.40 LDRSB (register) on page D3-922</a>
LDRSH (immediate)	Load Register Signed Halfword (immediate)	<a href="#">D3.41 LDRSH (immediate) on page D3-923</a>
LDRSH (register)	Load Register Signed Halfword (register)	<a href="#">D3.42 LDRSH (register) on page D3-924</a>
LDRSW (immediate)	Load Register Signed Word (immediate)	<a href="#">D3.43 LDRSW (immediate) on page D3-925</a>
LDRSW (literal)	Load Register Signed Word (literal)	<a href="#">D3.44 LDRSW (literal) on page D3-926</a>
LDRSW (register)	Load Register Signed Word (register)	<a href="#">D3.45 LDRSW (register) on page D3-927</a>
LDSETA, LDSETAL, LDSET, LDSETL, LDSETAL, LDSET, LDSETL	Atomic bit set on word or doubleword in memory	<a href="#">D3.46 LDSETA, LDSETAL, LDSET, LDSETL, LDSETAL, LDSET, LDSETL on page D3-928</a>
LDSETAB, LDSETALB, LDSETB, LDSETLB	Atomic bit set on byte in memory	<a href="#">D3.47 LDSETAB, LDSETALB, LDSETB, LDSETLB on page D3-929</a>
LDSETAH, LDSETALH, LDSETH, LDSETLH	Atomic bit set on halfword in memory	<a href="#">D3.48 LDSETAH, LDSETALH, LDSETH, LDSETLH on page D3-930</a>

**Table D3-1 Summary of A64 data transfer instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
LDSMAXA, LDSMAXAL, LDSMAX, LDSMAXL, LDSMAXAL, LDSMAX, LDSMAXL	Atomic signed maximum on word or doubleword in memory	<i>D3.49 LDSMAXA, LDSMAXAL, LDSMAX, LDSMAXL, LDSMAXAL, LDSMAX, LDSMAXL</i> on page D3-931
LDSMAXAB, LDSMAXALB, LDSMAXB, LDSMAXLB	Atomic signed maximum on byte in memory	<i>D3.50 LDSMAXAB, LDSMAXALB, LDSMAXB, LDSMAXLB</i> on page D3-932
LDSMAXAH, LDSMAXALH, LDSMAXH, LDSMAXLH	Atomic signed maximum on halfword in memory	<i>D3.51 LDSMAXAH, LDSMAXALH, LDSMAXH, LDSMAXLH</i> on page D3-933
LDSMINA, LDSMINAL, LDSMIN, LDSMINL, LDSMINAL, LDSMIN, LDSMINL	Atomic signed minimum on word or doubleword in memory	<i>D3.52 LDSMINA, LDSMINAL, LDSMIN, LDSMINL, LDSMINAL, LDSMIN, LDSMINL</i> on page D3-934
LDSMINAB, LDSMINALB, LDSMINB, LDSMINLB	Atomic signed minimum on byte in memory	<i>D3.53 LDSMINAB, LDSMINALB, LDSMINB, LDSMINLB</i> on page D3-935
LDSMINAH, LDSMINALH, LDSMINH, LDSMINLH	Atomic signed minimum on halfword in memory	<i>D3.54 LDSMINAH, LDSMINALH, LDSMINH, LDSMINLH</i> on page D3-936
LDTR	Load Register (unprivileged)	<i>D3.55 LDTR</i> on page D3-937
LDTRB	Load Register Byte (unprivileged)	<i>D3.56 LDTRB</i> on page D3-938
LDTRH	Load Register Halfword (unprivileged)	<i>D3.57 LDTRH</i> on page D3-939
LDTRSB	Load Register Signed Byte (unprivileged)	<i>D3.58 LDTRSB</i> on page D3-940
LDTRSH	Load Register Signed Halfword (unprivileged)	<i>D3.59 LDTRSH</i> on page D3-941
LDTRSW	Load Register Signed Word (unprivileged)	<i>D3.60 LDTRSW</i> on page D3-942
LDUMAXA, LDUMAXAL, LDUMAX, LDUMAXL, LDUMAXAL, LDUMAX, LDUMAXL	Atomic unsigned maximum on word or doubleword in memory	<i>D3.61 LDUMAXA, LDUMAXAL, LDUMAX, LDUMAXL, LDUMAXAL, LDUMAX, LDUMAXL</i> on page D3-943
LDUMAXAB, LDUMAXALB, LDUMAXB, LDUMAXLB	Atomic unsigned maximum on byte in memory	<i>D3.62 LDUMAXAB, LDUMAXALB, LDUMAXB, LDUMAXLB</i> on page D3-944
LDUMAXAH, LDUMAXALH, LDUMAXH, LDUMAXLH	Atomic unsigned maximum on halfword in memory	<i>D3.63 LDUMAXAH, LDUMAXALH, LDUMAXH, LDUMAXLH</i> on page D3-945
LDUMINA, LDUMINAL, LDUMIN, LDUMINL, LDUMINAL, LDUMIN, LDUMINL	Atomic unsigned minimum on word or doubleword in memory	<i>D3.64 LDUMINA, LDUMINAL, LDUMIN, LDUMINL, LDUMINAL, LDUMIN, LDUMINL</i> on page D3-946
LDUMINAB, LDUMINALB, LDUMINB, LDUMINLB	Atomic unsigned minimum on byte in memory	<i>D3.65 LDUMINAB, LDUMINALB, LDUMINB, LDUMINLB</i> on page D3-947
LDUMINAH, LDUMINALH, LDUMINH, LDUMINLH	Atomic unsigned minimum on halfword in memory	<i>D3.66 LDUMINAH, LDUMINALH, LDUMINH, LDUMINLH</i> on page D3-948
LDUR	Load Register (unscaled)	<i>D3.67 LDUR</i> on page D3-949
LDURB	Load Register Byte (unscaled)	<i>D3.68 LDURB</i> on page D3-950

**Table D3-1 Summary of A64 data transfer instructions (continued)**

Mnemonic	Brief description	See
LDURH	Load Register Halfword (unscaled)	<a href="#">D3.69 LDURH on page D3-951</a>
LDURSB	Load Register Signed Byte (unscaled)	<a href="#">D3.70 LDURSB on page D3-952</a>
LDURSH	Load Register Signed Halfword (unscaled)	<a href="#">D3.71 LDURSH on page D3-953</a>
LDURSW	Load Register Signed Word (unscaled)	<a href="#">D3.72 LDURSW on page D3-954</a>
LDXP	Load Exclusive Pair of Registers	<a href="#">D3.73 LDXP on page D3-955</a>
LDXR	Load Exclusive Register	<a href="#">D3.74 LDXR on page D3-956</a>
LDXRB	Load Exclusive Register Byte	<a href="#">D3.75 LDXRB on page D3-957</a>
LDXRH	Load Exclusive Register Halfword	<a href="#">D3.76 LDXRH on page D3-958</a>
PRFM (immediate)	Prefetch Memory (immediate)	<a href="#">D3.77 PRFM (immediate) on page D3-959</a>
PRFM (literal)	Prefetch Memory (literal)	<a href="#">D3.78 PRFM (literal) on page D3-961</a>
PRFM (register)	Prefetch Memory (register)	<a href="#">D3.79 PRFM (register) on page D3-963</a>
PRFUM (unscaled offset)	Prefetch Memory (unscaled offset)	<a href="#">D3.80 PRFUM (unscaled offset) on page D3-965</a>
STADD, STADDL, STADDL	Atomic add on word or doubleword in memory, without return	<a href="#">D3.81 STADD, STADDL, STADDL on page D3-967</a>
STADDB, STADDLB	Atomic add on byte in memory, without return	<a href="#">D3.82 STADDB, STADDLB on page D3-968</a>
STADDH, STADDLH	Atomic add on halfword in memory, without return	<a href="#">D3.83 STADDH, STADDLH on page D3-969</a>
STCLR, STCLRL, STCLRL	Atomic bit clear on word or doubleword in memory, without return	<a href="#">D3.84 STCLR, STCLRL, STCLRL on page D3-970</a>
STCLRB, STCLRLB	Atomic bit clear on byte in memory, without return	<a href="#">D3.85 STCLRB, STCLRLB on page D3-971</a>
STCLRH, STCLRLH	Atomic bit clear on halfword in memory, without return	<a href="#">D3.86 STCLRH, STCLRLH on page D3-972</a>
STEOR, STEORL, STEORL	Atomic exclusive OR on word or doubleword in memory, without return	<a href="#">D3.87 STEOR, STEORL, STEORL on page D3-973</a>
STEORB, STEORLB	Atomic exclusive OR on byte in memory, without return	<a href="#">D3.88 STEORB, STEORLB on page D3-974</a>
STEORH, STEORLH	Atomic exclusive OR on halfword in memory, without return	<a href="#">D3.89 STEORH, STEORLH on page D3-975</a>
STLLR	Store LORelease Register	<a href="#">D3.90 STLLR on page D3-976</a>
STLLRB	Store LORelease Register Byte	<a href="#">D3.91 STLLRB on page D3-977</a>
STLLRH	Store LORelease Register Halfword	<a href="#">D3.92 STLLRH on page D3-978</a>
STLR	Store-Release Register	<a href="#">D3.93 STLR on page D3-979</a>
STLRB	Store-Release Register Byte	<a href="#">D3.94 STLRB on page D3-980</a>
STLRH	Store-Release Register Halfword	<a href="#">D3.95 STLRH on page D3-981</a>
STLXP	Store-Release Exclusive Pair of registers	<a href="#">D3.96 STLXP on page D3-982</a>
STLXR	Store-Release Exclusive Register	<a href="#">D3.97 STLXR on page D3-984</a>

**Table D3-1 Summary of A64 data transfer instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
STLXR <sub>B</sub>	Store-Release Exclusive Register Byte	<a href="#">D3.98 STLXR<sub>B</sub> on page D3-986</a>
STLXR <sub>H</sub>	Store-Release Exclusive Register Halfword	<a href="#">D3.99 STLXR<sub>H</sub> on page D3-987</a>
STNP	Store Pair of Registers, with non-temporal hint	<a href="#">D3.100 STNP on page D3-988</a>
STP	Store Pair of Registers	<a href="#">D3.101 STP on page D3-989</a>
STR (immediate)	Store Register (immediate)	<a href="#">D3.102 STR (immediate) on page D3-990</a>
STR (register)	Store Register (register)	<a href="#">D3.103 STR (register) on page D3-991</a>
STRB (immediate)	Store Register Byte (immediate)	<a href="#">D3.104 STRB (immediate) on page D3-992</a>
STRB (register)	Store Register Byte (register)	<a href="#">D3.105 STRB (register) on page D3-993</a>
STRH (immediate)	Store Register Halfword (immediate)	<a href="#">D3.106 STRH (immediate) on page D3-994</a>
STRH (register)	Store Register Halfword (register)	<a href="#">D3.107 STRH (register) on page D3-995</a>
STSET, STSETL, STSETL <sub>B</sub>	Atomic bit set on word or doubleword in memory, without return	<a href="#">D3.108 STSET, STSETL, STSETL<sub>B</sub> on page D3-996</a>
STSETB, STSETLB	Atomic bit set on byte in memory, without return	<a href="#">D3.109 STSETB, STSETLB on page D3-997</a>
STSETH, STSETL <sub>H</sub>	Atomic bit set on halfword in memory, without return	<a href="#">D3.110 STSETH, STSETL<sub>H</sub> on page D3-998</a>
STSMAX, STSMAXL, STSMAXL <sub>B</sub>	Atomic signed maximum on word or doubleword in memory, without return	<a href="#">D3.111 STSMAX, STSMAXL, STSMAXL<sub>B</sub> on page D3-999</a>
STSMAXB, STSMAXLB	Atomic signed maximum on byte in memory, without return	<a href="#">D3.112 STSMAXB, STSMAXLB on page D3-1000</a>
STSMAXH, STSMAXL <sub>H</sub>	Atomic signed maximum on halfword in memory, without return	<a href="#">D3.113 STSMAXH, STSMAXL<sub>H</sub> on page D3-1001</a>
STSMIN, STSMINL, STSMINL <sub>B</sub>	Atomic signed minimum on word or doubleword in memory, without return	<a href="#">D3.114 STSMIN, STSMINL, STSMINL<sub>B</sub> on page D3-1002</a>
STSMINB, STSMINLB	Atomic signed minimum on byte in memory, without return	<a href="#">D3.115 STSMINB, STSMINLB on page D3-1003</a>
STSMINH, STSMINL <sub>H</sub>	Atomic signed minimum on halfword in memory, without return	<a href="#">D3.116 STSMINH, STSMINL<sub>H</sub> on page D3-1004</a>
STTR	Store Register (unprivileged)	<a href="#">D3.117 STTR on page D3-1005</a>
STTRB	Store Register Byte (unprivileged)	<a href="#">D3.118 STTRB on page D3-1006</a>
STTRH	Store Register Halfword (unprivileged)	<a href="#">D3.119 STTRH on page D3-1007</a>
STUMAX, STUMAXL, STUMAXL <sub>B</sub>	Atomic unsigned maximum on word or doubleword in memory, without return	<a href="#">D3.120 STUMAX, STUMAXL, STUMAXL<sub>B</sub> on page D3-1008</a>
STUMAXB, STUMAXLB	Atomic unsigned maximum on byte in memory, without return	<a href="#">D3.121 STUMAXB, STUMAXLB on page D3-1009</a>
STUMAXH, STUMAXL <sub>H</sub>	Atomic unsigned maximum on halfword in memory, without return	<a href="#">D3.122 STUMAXH, STUMAXL<sub>H</sub> on page D3-1010</a>

**Table D3-1 Summary of A64 data transfer instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
STUMIN, STUMINL, STUMINL	Atomic unsigned minimum on word or doubleword in memory, without return	<a href="#">D3.123 STUMIN, STUMINL, STUMINL on page D3-1011</a>
STUMINB, STUMINLB	Atomic unsigned minimum on byte in memory, without return	<a href="#">D3.124 STUMINB, STUMINLB on page D3-1012</a>
STUMINH, STUMINLH	Atomic unsigned minimum on halfword in memory, without return	<a href="#">D3.125 STUMINH, STUMINLH on page D3-1013</a>
STUR	Store Register (unscaled)	<a href="#">D3.126 STUR on page D3-1014</a>
STURB	Store Register Byte (unscaled)	<a href="#">D3.127 STURB on page D3-1015</a>
STURH	Store Register Halfword (unscaled)	<a href="#">D3.128 STURH on page D3-1016</a>
STXP	Store Exclusive Pair of registers	<a href="#">D3.129 STXP on page D3-1017</a>
STXR	Store Exclusive Register	<a href="#">D3.130 STXR on page D3-1019</a>
STXRB	Store Exclusive Register Byte	<a href="#">D3.131 STXRB on page D3-1021</a>
STXRH	Store Exclusive Register Halfword	<a href="#">D3.132 STXRH on page D3-1022</a>
SWPA, SWPAL, SWP, SWPL, SWPAL, SWP, SWPL	Swap word or doubleword in memory	<a href="#">D3.133 SWPA, SWPAL, SWP, SWPL, SWPAL, SWP, SWPL on page D3-1023</a>
SWPAB, SWPALB, SWPB, SWPLB	Swap byte in memory	<a href="#">D3.134 SWPAB, SWPALB, SWPB, SWPLB on page D3-1024</a>
SWPAH, SWPALH, SWPH, SWPLH	Swap halfword in memory	<a href="#">D3.135 SWPAH, SWPALH, SWPH, SWPLH on page D3-1025</a>

## D3.2 CASA, CASAL, CAS, CASL, CASAL, CAS, CASL

Compare and Swap word or doubleword in memory.

### Syntax

```
CASA Ws, Wt, [Xn/SP{,#0}] ; 32-bit, acquire
CASAL Ws, Wt, [Xn/SP{,#0}] ; 32-bit, acquire and release
CAS Ws, Wt, [Xn/SP{,#0}] ; 32-bit, no memory ordering
CASL Ws, Wt, [Xn/SP{,#0}] ; 32-bit, release
CASA Xs, Xt, [Xn/SP{,#0}] ; 64-bit, acquire
CASAL Xs, Xt, [Xn/SP{,#0}] ; 64-bit, acquire and release
CAS Xs, Xt, [Xn/SP{,#0}] ; 64-bit, no memory ordering
CASL Xs, Xt, [Xn/SP{,#0}] ; 64-bit, release
```

Where:

*Ws*

Is the 32-bit name of the general-purpose register to be compared and loaded.

*Wt*

Is the 32-bit name of the general-purpose register to be conditionally stored.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Xs*

Is the 64-bit name of the general-purpose register to be compared and loaded.

*Xt*

Is the 64-bit name of the general-purpose register to be conditionally stored.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword 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.

- CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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*, or *Xs*, is restored to the value held in the register before the instruction was executed.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.3 CASAB, CASALB, CASB, CASLB

Compare and Swap byte in memory.

### Syntax

```
CASAB Ws, Wt, [Xn/SP{,#0}] ; Acquire general registers  
CASALB Ws, Wt, [Xn/SP{,#0}] ; Acquire and release general registers  
CASB Ws, Wt, [Xn/SP{,#0}] ; No memory ordering general registers  
CASLB Ws, Wt, [Xn/SP{,#0}] ; Release general registers
```

Where:

*Ws*

Is the 32-bit name of the general-purpose register to be compared and loaded.

*Wt*

Is the 32-bit name of the general-purpose register to be conditionally stored.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.4 CASAH, CASALH, CASH, CASLH

Compare and Swap halfword in memory.

### Syntax

```
CASAH Ws, Wt, [Xn/SP{,#0}] ; Acquire general registers  
CASALH Ws, Wt, [Xn/SP{,#0}] ; Acquire and release general registers  
CASH Ws, Wt, [Xn/SP{,#0}] ; No memory ordering general registers  
CASLH Ws, Wt, [Xn/SP{,#0}] ; Release general registers
```

Where:

*Ws*

Is the 32-bit name of the general-purpose register to be compared and loaded.

*Wt*

Is the 32-bit name of the general-purpose register to be conditionally stored.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.5 CASPA, CASPAL, CASP, CASPL, CASPAL, CASP, CASPL

Compare and Swap Pair of words or doublewords in memory.

### Syntax

```
CASPA ws, <W(s+1)>, wt, <W(t+1)>, [Xn/SP{,#0}] ; 32-bit, acquire
CASPAL ws, <W(s+1)>, wt, <W(t+1)>, [Xn/SP{,#0}] ; 32-bit, acquire and release
CASP ws, <W(s+1)>, wt, <W(t+1)>, [Xn/SP{,#0}] ; 32-bit, no memory ordering
CASPL ws, <W(s+1)>, wt, <W(t+1)>, [Xn/SP{,#0}] ; 32-bit, release
CASPA xs, <X(s+1)>, xt, <X(t+1)>, [Xn/SP{,#0}] ; 64-bit, acquire
CASPAL xs, <X(s+1)>, xt, <X(t+1)>, [Xn/SP{,#0}] ; 64-bit, acquire and release
CASP xs, <X(s+1)>, xt, <X(t+1)>, [Xn/SP{,#0}] ; 64-bit, no memory ordering
CASPL xs, <X(s+1)>, xt, <X(t+1)>, [Xn/SP{,#0}] ; 64-bit, release
```

Where:

**ws**

Is the 32-bit name of the first general-purpose register to be compared and loaded.

**<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.

**<W(t+1)>**

Is the 32-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.

**xs**

Is the 64-bit name of the first general-purpose register to be compared and loaded.

**<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.

**<X(t+1)>**

Is the 64-bit name of the second general-purpose register to be conditionally stored.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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  $\langle W(s+1) \rangle$ , or  $Xs$  and  $\langle X(s+1) \rangle$ , are restored to the values held in the registers before the instruction was executed.

**Related reference**

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.6 LDADDA, LDADDAL, LDADD, LDADDL, LDADDAL, LDADD, LDADDL

Atomic add on word or doubleword in memory.

### Syntax

```
LDADDA Ws, Wt, [Xn/SP] ; 32-bit, acquire
LDADDAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release
LDADD Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering
LDADDL Ws, Wt, [Xn/SP] ; 32-bit, release
LDADDA Xs, Xt, [Xn/SP] ; 64-bit, acquire
LDADDAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release
LDADD Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering
LDADDL Xs, Xt, [Xn/SP] ; 64-bit, release
```

Where:

**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.

**Wt**

Is the 32-bit name of the general-purpose register to be loaded.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**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.

**Xt**

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.7 LDADDAB, LDADDALB, LDADDB, LDADDLB

Atomic add on byte in memory.

### Syntax

```
LDADDAB Ws, Wt, [Xn/SP] ; Acquire general registers  
LDADDALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDADDB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDADDLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.8 LDADDAAH, LDADDALH, LDADDH, LDADDLH

Atomic add on halfword in memory.

### Syntax

```
LDADDAAH Ws, Wt, [Xn/SP] ; Acquire general registers  
LDADDALH Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDADDH Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDADDLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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, LDADDAAH 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.9 LDAPR

Load-Acquire RCpc Register.

### Syntax

LDAPR *Wt*, [*Xn/SP* {,#0}] ; 32-bit

LDAPR *Xt*, [*Xn/SP* {,#0}] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xt*

Is the 64-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

This instruction is supported in the Armv8.3-A architecture and later. It is optionally supported in the Armv8.2-A architecture with the RCpc extension.

### Usage

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.10 LDAPRB

Load-Acquire RCpc Register Byte.

### Syntax

LDAPRB *Wt*, [*Xn/SP* {,#0}]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

This instruction is supported in the Armv8.3-A architecture and later. It is optionally supported in the Armv8.2-A architecture with the RCpc extension.

### Usage

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.11 LDAPRH

Load-Acquire RCpc Register Halfword.

### Syntax

LDAPRH *Wt*, [*Xn/SP* {,#0}]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

This instruction is supported in the Armv8.3-A architecture and later. It is optionally supported in the Armv8.2-A architecture with the RCpc extension.

### Usage

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.12 LDAR

Load-Acquire Register.

### Syntax

LDAR *Wt*, [*Xn/SP{,#0}*] ; 32-bit

LDAR *Xt*, [*Xn/SP{,#0}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.13 LDARB

Load-Acquire Register Byte.

### Syntax

LDARB *Wt*, [*Xn/SP{, #0}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.14 LDARH

Load-Acquire Register Halfword.

### Syntax

LDARH *Wt*, [*Xn/SP{, #0}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Armv8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.15 LDAXP

Load-Acquire Exclusive Pair of Registers.

### Syntax

LDAXP *Wt1*, *Wt2*, [*Xn/SP{, #0}*] ; 32-bit

LDAXP *Xt1*, *Xt2*, [*Xn/SP{, #0}*] ; 64-bit

Where:

***Wt1***

Is the 32-bit name of the first general-purpose register to be transferred.

***Wt2***

Is the 32-bit name of the second general-purpose register to be transferred.

***Xt1***

Is the 64-bit name of the first general-purpose register to be transferred.

***Xt2***

Is the 64-bit name of the second general-purpose register to be transferred.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly LDAXP.

---

#### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

#### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.16 LDAXR

Load-Acquire Exclusive Register.

### Syntax

LDAXR *Wt*, [*Xn/SP{,#0}*] ; 32-bit

LDAXR *Xt*, [*Xn/SP{,#0}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.17 LDAXRB

Load-Acquire Exclusive Register Byte.

### Syntax

LDAXRB *Wt*, [*Xn/SP{, #0}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.18 LDAXRH

Load-Acquire Exclusive Register Halfword.

### Syntax

LDAXRH *Wt*, [Xn/SP{,#0}]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.19 LDCLRA, LDCLRAL, LDCLR, LDCLRL, LDCLRAL, LDCLR, LDCLRL

Atomic bit clear on word or doubleword in memory.

### Syntax

```
LDCLRA Ws, Wt, [Xn/SP] ; 32-bit, acquire  
LDCLRAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release  
LDCLR Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering  
LDCLRL Ws, Wt, [Xn/SP] ; 32-bit, release  
LDCLRA Xs, Xt, [Xn/SP] ; 64-bit, acquire  
LDCLRAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release  
LDCLR Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering  
LDCLRL Xs, Xt, [Xn/SP] ; 64-bit, release
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

*Xt*

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

For information about memory accesses see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.20 LDCLRAB, LDCLRALB, LDCLRAB, LDCLRLB

Atomic bit clear on byte in memory.

### Syntax

```
LDCLRAB Ws, Wt, [Xn/SP] ; Acquire general registers  
LDCLRALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDCLRAB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDCLRLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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 LDCLRALB load from memory with acquire semantics.
- LDCLRAB and LDCLRALB store to memory with release semantics.
- LDCLRAB has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.21 LDCLRAH, LDCLRALH, LDCLRH, LDCLRLH

Atomic bit clear on halfword in memory.

### Syntax

```
LDCLRAH Ws, Wt, [Xn/SP] ; Acquire general registers  
LDCLRALH Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDCLRH Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDCLRLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- LDCLRH has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.22 LDEORA, LDEORAL, LDEOR, LDEORL, LDEORAL, LDEOR, LDEORL

Atomic exclusive OR on word or doubleword in memory.

### Syntax

```
LDEORA Ws, Wt, [Xn/SP] ; 32-bit, acquire  
LDEORAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release  
LDEOR Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering  
LDEORL Ws, Wt, [Xn/SP] ; 32-bit, release  
LDEORA Xs, Xt, [Xn/SP] ; 64-bit, acquire  
LDEORAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release  
LDEOR Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering  
LDEORL Xs, Xt, [Xn/SP] ; 64-bit, release
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

*Xt*

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.23 LDEORAB, LDEORALB, LDEORB, LDEORLB

Atomic exclusive OR on byte in memory.

### Syntax

```
LDEORAB Ws, Wt, [Xn/SP] ; Acquire general registers  
LDEORALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDEORB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDEORLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.24 LDEORAH, LDEORALH, LDEORH, LDEORLH

Atomic exclusive OR on halfword in memory.

### Syntax

```
LDEORAH Ws, Wt, [Xn/SP] ; Acquire general registers  
LDEORALH Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDEORH Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDEORLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.25 LDLAR

Load LOAcquire Register.

### Syntax

LDLAR *Wt*, [*Xn/SP{,#0}*] ; 32-bit

LDLAR *Xt*, [*Xn/SP{,#0}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.26 LDLARB

Load LOAcquire Register Byte.

### Syntax

LDLARB *Wt*, [*Xn/SP{,#0}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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 LOResume. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.27 LDLARH

Load LOAcquire Register Halfword.

### Syntax

LDLARH *Wt*, [Xn/SP{,#0}]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

Xn/SP

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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 LOResume. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.28 LDNP

Load Pair of Registers, with non-temporal hint.

### Syntax

LDNP *Wt1*, *Wt2*, [*Xn/SP{, #imm}*] ; 32-bit

LDNP *Xt1*, *Xt2*, [*Xn/SP{, #imm}*] ; 64-bit

Where:

*Wt1*

Is the 32-bit name of the first general-purpose register to be transferred.

*Wt2*

Is the 32-bit name of the second general-purpose register to be transferred.

*imm*

Depends on the instruction variant:

#### 32-bit general registers

Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

#### 64-bit general registers

Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0.

*Xt1*

Is the 64-bit name of the first general-purpose register to be transferred.

*Xt2*

Is the 64-bit name of the second general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Note

---

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly LDNP.

---

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.29 LDP

Load Pair of Registers.

### Syntax

```
LDP Wt1, Wt2, [Xn/SP], #imm ; 32-bit
LDP Xt1, Xt2, [Xn/SP], #imm ; 64-bit
LDP Wt1, Wt2, [Xn/SP, #imm]! ; 32-bit
LDP Xt1, Xt2, [Xn/SP, #imm]! ; 64-bit
LDP Wt1, Wt2, [Xn/SP{, #imm}] ; 32-bit
LDP Xt1, Xt2, [Xn/SP{, #imm}] ; 64-bit
```

Where:

**Wt1**

Is the 32-bit name of the first general-purpose register to be transferred.

**Wt2**

Is the 32-bit name of the second general-purpose register to be transferred.

**imm**

Depends on the instruction variant:

#### 32-bit general registers

Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

#### 64-bit general registers

Is the signed immediate byte offset, a multiple of 8 in the range -512 to 504.

**Xt1**

Is the 64-bit name of the first general-purpose register to be transferred.

**Xt2**

Is the 64-bit name of the second general-purpose register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

#### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDP.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.30 LDPSW

Load Pair of Registers Signed Word.

### Syntax

```
LDPSW Xt1, Xt2, [Xn/SP], #imm ; Post-index general registers  
LDPSW Xt1, Xt2, [Xn/SP, #imm]! ; Pre-index general registers  
LDPSW Xt1, Xt2, [Xn/SP{, #imm}] ; Signed offset general registers
```

Where:

*imm*

Depends on the instruction variant:

#### Post-index and Pre-index general registers

Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

#### Signed offset general registers

Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

**Xt1**

Is the 64-bit name of the first general-purpose register to be transferred.

**Xt2**

Is the 64-bit name of the second general-purpose register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Note

---

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly LDPSW.

---

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.31 LDR (immediate)

Load Register (immediate).

### Syntax

```
LDR Wt, [Xn/SP], #simm ; 32-bit  
LDR Xt, [Xn/SP], #simm ; 64-bit  
LDR Wt, [Xn/SP, #simm]! ; 32-bit  
LDR Xt, [Xn/SP, #simm]! ; 64-bit  
LDR Wt, [Xn/SP{, #pimm}] ; 32-bit  
LDR Xt, [Xn/SP{, #pimm}] ; 64-bit
```

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*simm*

Is the signed immediate byte offset, in the range -256 to 255.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*pimm*

Depends on the instruction variant:

#### 32-bit general registers

Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

#### 64-bit general registers

Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

---

#### Note

---

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly LDR (immediate).

---

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.32 LDR (literal)

Load Register (literal).

### Syntax

LDR *Wt*, *Label* ; 32-bit

LDR *Xt*, *Label* ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xt*

Is the 64-bit name of the general-purpose register to be loaded.

*Label*

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range  $\pm 1\text{MB}$ .

### Usage

Load Register (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* in the [Arm® Architecture Reference Manual Armv8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.33 LDR (register)

Load Register (register).

### Syntax

LDR *Wt*, [*Xn/SP*, (*Wm|Xm*){}*, extend {amount}*{}]] ; 32-bit

LDR *Xt*, [*Xn/SP*, (*Wm|Xm*){}*, extend {amount}*{}]] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*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:

#### 32-bit general registers

Can be one of #0 or #2.

#### 64-bit general registers

Can be one of #0 or #3.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Wm*

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

*Xm*

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

*extend*

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of the values shown in Usage.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.34 LDRAA, LDRAB, LDRAB

Load Register, with pointer authentication.

### Syntax

```
LDRAA Xt, [Xn/SP{, #simm}] ; LDRAA  
LDRAA Xt, [Xn/SP{, #simm}]! ; LDRAA  
LDRAB Xt, [Xn/SP{, #simm}] ; LDRAB  
LDRAB Xt, [Xn/SP{, #simm}]! ; LDRAB
```

Where:

**Xt**

Is the 64-bit name of the general-purpose register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**simm**

Is the optional signed immediate byte offset, in the range -512 to 511, defaulting to 0.

### Usage

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for LDRAA, and key B is used for LDRAB.

If the authentication passes, the PE behaves the same as for an LDR instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.35 LDRB (immediate)

Load Register Byte (immediate).

### Syntax

```
LDRB Wt, [Xn/SP], #simm ; Post-index general registers  
LDRB Wt, [Xn/SP, #simm]! ; Pre-index general registers  
LDRB Wt, [Xn/SP{, #pimm}] ; Unsigned offset general registers
```

Where:

*simm*

Is the signed immediate byte offset, in the range -256 to 255.

*pimm*

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly LDRH (immediate).

---

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.36 LDRB (register)

Load Register Byte (register).

### Syntax

LDRB *Wt*, [Xn/SP, (*Wm|Xm*), *extend {amount}*] ; Extended register general registers

LDRB *Wt*, [Xn/SP, *Xm{, LSL amount}*] ; Shifted register general registers

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Wm*

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

*Xm*

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

*extend*

Is the index extend specifier, and can be one of the values shown in Usage.

*amount*

Is the index shift amount, it must be.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.37 LDRH (immediate)

Load Register Halfword (immediate).

### Syntax

```
LDRH Wt, [Xn/SP], #simm ; Post-index general registers  
LDRH Wt, [Xn/SP, #simm]! ; Pre-index general registers  
LDRH Wt, [Xn/SP{, #pimm}] ; Unsigned offset general registers
```

Where:

*simm*

Is the signed immediate byte offset, in the range -256 to 255.

*pimm*

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

#### Note

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDRH (immediate).

---

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.38 LDRH (register)

Load Register Halfword (register).

### Syntax

LDRH *Wt*, [Xn/SP, (*Wm*|*Xm*) {, *extend* {*amount*} }]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Wm*

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

*Xm*

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

*extend*

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of UXTW, LSL, SXTW or 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, and can be either #0 or #1.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDRH (register).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.39 LDRSB (immediate)

Load Register Signed Byte (immediate).

### Syntax

```
LDRSB Wt, [Xn/SP], #simm ; 32-bit  
LDRSB Xt, [Xn/SP], #simm ; 64-bit  
LDRSB Wt, [Xn/SP, #simm]! ; 32-bit  
LDRSB Xt, [Xn/SP, #simm]! ; 64-bit  
LDRSB Wt, [Xn/SP{, #pimm}] ; 32-bit  
LDRSB Xt, [Xn/SP{, #pimm}] ; 64-bit
```

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*simm*

Is the signed immediate byte offset, in the range -256 to 255.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*pimm*

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

#### Note

---

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDRSB (immediate).

---

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.40 LDRSB (register)

Load Register Signed Byte (register).

### Syntax

LDRSB *Wt*, [*Xn/SP*, (*Wm|Xm*), *extend {amount}*] ; 32-bit with extended register offset

LDRSB *Wt*, [*Xn/SP*, *Xm{, LSL amount}*] ; 32-bit with shifted register offset

LDRSB *Xt*, [*Xn/SP*, (*Wm|Xm*), *extend {amount}*] ; 64-bit with extended register offset

LDRSB *Xt*, [*Xn/SP*, *Xm{, LSL amount}*] ; 64-bit with shifted register offset

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Wm*

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

*Xm*

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

*extend*

Is the index extend specifier.

Can be one of UXTW, SXTW or SXTX.

*amount*

Is the index shift amount, it must be.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.41 LDRSH (immediate)

Load Register Signed Halfword (immediate).

### Syntax

```
LDRSH Wt, [Xn/SP], #simm ; 32-bit  
LDRSH Xt, [Xn/SP], #simm ; 64-bit  
LDRSH Wt, [Xn/SP, #simm]! ; 32-bit  
LDRSH Xt, [Xn/SP, #simm]! ; 64-bit  
LDRSH Wt, [Xn/SP{, #pimm}] ; 32-bit  
LDRSH Xt, [Xn/SP{, #pimm}] ; 64-bit
```

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*simm*

Is the signed immediate byte offset, in the range -256 to 255.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*pimm*

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

#### Note

---

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDRSH (immediate).

---

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.42 LDRSH (register)

Load Register Signed Halfword (register).

### Syntax

LDRSH *Wt*, [*Xn/SP*, (*Wm|Xm*){}, *extend* {*amount*}] ; 32-bit

LDRSH *Xt*, [*Xn/SP*, (*Wm|Xm*){}, *extend* {*amount*}] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Wm*

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

*Xm*

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

*extend*

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of the values shown in Usage.

*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, and can be either #0 or #1.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.43 LDRSW (immediate)

Load Register Signed Word (immediate).

### Syntax

```
LDRSW Xt, [Xn/SP], #simm ; Post-index general registers  
LDRSW Xt, [Xn/SP, #simm]! ; Pre-index general registers  
LDRSW Xt, [Xn/SP{, #pimm}] ; Unsigned offset general registers
```

Where:

**simm**

Is the signed immediate byte offset, in the range -256 to 255.

**pimm**

Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

**Xt**

Is the 64-bit name of the general-purpose register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

#### Note

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDRSW (immediate).

#### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

#### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.44 LDRSW (literal)

Load Register Signed Word (literal).

### Syntax

LDRSW *Xt*, *Label*

Where:

**Xt**

Is the 64-bit name of the general-purpose register to be loaded.

**Label**

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range  $\pm 1\text{MB}$ .

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.45 LDRSW (register)

Load Register Signed Word (register).

### Syntax

LDRSW *Xt*, [*Xn/SP*, (*Wm|Xm*){}, *extend* {*amount*}{}]

Where:

***Xt***

Is the 64-bit name of the general-purpose register to be transferred.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

***Wm***

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

***Xm***

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

***extend***

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of UXTW, LSL, SXTW or 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, and can be either #0 or #2.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.46 LDSETA, LDSETAL, LDSET, LDSETL, LDSETAL, LDSET, LDSETL

Atomic bit set on word or doubleword in memory.

### Syntax

```
LDSETA Ws, Wt, [Xn/SP] ; 32-bit, acquire
LDSETAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release
LDSET Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering
LDSETL Ws, Wt, [Xn/SP] ; 32-bit, release
LDSETA Xs, Xt, [Xn/SP] ; 64-bit, acquire
LDSETAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release
LDSET Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering
LDSETL Xs, Xt, [Xn/SP] ; 64-bit, release
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

*Xt*

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.47 LDSETAB, LDSETALB, LDSETB, LDSETLB

Atomic bit set on byte in memory.

### Syntax

```
LDSETAB Ws, Wt, [Xn/SP] ; Acquire general registers  
LDSETALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDSETB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDSETLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.48 LDSETAH, LDSETALH, LDSETH, LDSETLH

Atomic bit set on halfword in memory.

### Syntax

```
LDSETAH Ws, Wt, [Xn/SP] ; Acquire general registers  
LDSETALH Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDSETH Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDSETLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.49 LDSMAXA, LDSMAXAL, LDSMAX, LDSMAXL, LDSMAXAL, LDSMAX, LDSMAXL

Atomic signed maximum on word or doubleword in memory.

### Syntax

```

LDSMAXA Ws, Wt, [Xn/SP] ; 32-bit, acquire
LDSMAXAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release
LDSMAX Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering
LDSMAXL Ws, Wt, [Xn/SP] ; 32-bit, release
LDSMAXA Xs, Xt, [Xn/SP] ; 64-bit, acquire
LDSMAXAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release
LDSMAX Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering
LDSMAXL Xs, Xt, [Xn/SP] ; 64-bit, release

```

Where:

**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.

**Wt**

Is the 32-bit name of the general-purpose register to be loaded.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**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.

**Xt**

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.50 LDSMAXAB, LDSMAXALB, LDSMAXB, LDSMAXLB

Atomic signed maximum on byte in memory.

### Syntax

```
LDSMAXAB Ws, Wt, [Xn/SP] ; Acquire general registers  
LDSMAXALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDSMAXB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDSMAXLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.51 LDSMAXAH, LDSMAXALH, LDSMAXH, LDSMAXLH

Atomic signed maximum on halfword in memory.

### Syntax

```
LDSMAXAH Ws, Wt, [Xn/SP] ; Acquire general registers  
LDSMAXALH Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDSMAXH Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDSMAXLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.52 LDSMINA, LDSMINAL, LDSMIN, LDSMINL, LDSMINAL, LDSMIN, LDSMINL

Atomic signed minimum on word or doubleword in memory.

### Syntax

```
LDSMINA Ws, Wt, [Xn/SP] ; 32-bit, acquire  
LDSMINAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release  
LDSMIN Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering  
LDSMINL Ws, Wt, [Xn/SP] ; 32-bit, release  
LDSMINA Xs, Xt, [Xn/SP] ; 64-bit, acquire  
LDSMINAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release  
LDSMIN Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering  
LDSMINL Xs, Xt, [Xn/SP] ; 64-bit, release
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

*Xt*

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

For information about memory accesses see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.53 LDSMINAB, LDSMINALB, LDSMINB, LDSMINLB

Atomic signed minimum on byte in memory.

### Syntax

```
LDSMINAB Ws, Wt, [Xn/SP] ; Acquire general registers  
LDSMINALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDSMINB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDSMINLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.54 LDSMINAH, LDSMINALH, LDSMINH, LDSMINLH

Atomic signed minimum on halfword in memory.

### Syntax

```
LDSMINAH Ws, Wt, [Xn/SP] ; Acquire general registers
LDSMINALH Ws, Wt, [Xn/SP] ; Acquire and release general registers
LDSMINH Ws, Wt, [Xn/SP] ; No memory ordering general registers
LDSMINLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

**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.

**Wt**

Is the 32-bit name of the general-purpose register to be loaded.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.55 LDTR

Load Register (unprivileged).

### Syntax

LDTR *Wt*, [*Xn/SP{, #simm}*] ; 32-bit

LDTR *Xt*, [*Xn/SP{, #simm}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.56 LDTRB

Load Register Byte (unprivileged).

### Syntax

LDTRB *Wt*, [*Xn/SP{, #simm}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.57 LDTRH

Load Register Halfword (unprivileged).

### Syntax

LDTRH *Wt*, [*Xn/SP*{, #*simm*}]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.58 LDTRSB

Load Register Signed Byte (unprivileged).

### Syntax

LDTRSB *Wt*, [*Xn/SP*{, #*simm*}] ; 32-bit

LDTRSB *Xt*, [*Xn/SP*{, #*simm*}] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.59 LDTRSH

Load Register Signed Halfword (unprivileged).

### Syntax

LDTRSH *Wt*, [*Xn/SP*{, #*simm*}] ; 32-bit

LDTRSH *Xt*, [*Xn/SP*{, #*simm*}] ; 64-bit

Where:

***Wt***

Is the 32-bit name of the general-purpose register to be transferred.

***Xt***

Is the 64-bit name of the general-purpose register to be transferred.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

***simm***

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.60 LDTRSW

Load Register Signed Word (unprivileged).

### Syntax

LDTRSW *Xt*, [*Xn/SP{*, #*simm**}*]

Where:

**Xt**

Is the 64-bit name of the general-purpose register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**simm**

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.61 LDUMAXA, LDUMAXAL, LDUMAX, LDUMAXL, LDUMAXAL, LDUMAX, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory.

### Syntax

```
LDUMAXA Ws, Wt, [Xn/SP] ; 32-bit, acquire  
LDUMAXAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release  
LDUMAX Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering  
LDUMAXL Ws, Wt, [Xn/SP] ; 32-bit, release  
LDUMAXA Xs, Xt, [Xn/SP] ; 64-bit, acquire  
LDUMAXAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release  
LDUMAX Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering  
LDUMAXL Xs, Xt, [Xn/SP] ; 64-bit, release
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

*Xt*

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.62 LDUMAXAB, LDUMAXALB, LDUMAXB, LDUMAXLB

Atomic unsigned maximum on byte in memory.

### Syntax

```
LDUMAXAB Ws, Wt, [Xn/SP] ; Acquire general registers  
LDUMAXALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDUMAXB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDUMAXLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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, LDUMAXAB 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.63 LDUMAXAH, LDUMAXALH, LDUMAXH, LDUMAXLH

Atomic unsigned maximum on halfword in memory.

### Syntax

```
LDUMAXAH Ws, Wt, [Xn/SP] ; Acquire general registers  
LDUMAXALH Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDUMAXH Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDUMAXLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.64 LDUMINA, LDUMINAL, LDUMIN, LDUMINL, LDUMINAL, LDUMIN, LDUMINL

Atomic unsigned minimum on word or doubleword in memory.

### Syntax

```
LDUMINA Ws, Wt, [Xn/SP] ; 32-bit, acquire
LDUMINAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release
LDUMIN Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering
LDUMINL Ws, Wt, [Xn/SP] ; 32-bit, release
LDUMINA Xs, Xt, [Xn/SP] ; 64-bit, acquire
LDUMINAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release
LDUMIN Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering
LDUMINL Xs, Xt, [Xn/SP] ; 64-bit, release
```

Where:

**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.

**Wt**

Is the 32-bit name of the general-purpose register to be loaded.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**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.

**Xt**

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

For information about memory accesses see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.65 LDUMINAB, LDUMINALB, LDUMINB, LDUMINLB

Atomic unsigned minimum on byte in memory.

### Syntax

```
LDUMINAB Ws, Wt, [Xn/SP] ; Acquire general registers  
LDUMINALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDUMINB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDUMINLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.66 LDUMINAH, LDUMINALH, LDUMINH, LDUMINLH

Atomic unsigned minimum on halfword in memory.

### Syntax

```
LDUMINAH Ws, Wt, [Xn/SP] ; Acquire general registers  
LDUMINALH Ws, Wt, [Xn/SP] ; Acquire and release general registers  
LDUMINH Ws, Wt, [Xn/SP] ; No memory ordering general registers  
LDUMINLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*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.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.67 LDUR

Load Register (unscaled).

### Syntax

LDUR *Wt*, [*Xn/SP{*, #*simm*}] ; 32-bit

LDUR *Xt*, [*Xn/SP{*, #*simm*}] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.68 LDURB

Load Register Byte (unscaled).

### Syntax

LDURB *Wt*, [*Xn/SP{*, *#simm**}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.69 LDURH

Load Register Halfword (unscaled).

### Syntax

LDURH *Wt*, [*Xn/SP{, #simm}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.70 LDURSB

Load Register Signed Byte (unscaled).

### Syntax

LDURSB *Wt*, [*Xn/SP{, #simm}*] ; 32-bit

LDURSB *Xt*, [*Xn/SP{, #simm}*] ; 64-bit

Where:

***Wt***

Is the 32-bit name of the general-purpose register to be transferred.

***Xt***

Is the 64-bit name of the general-purpose register to be transferred.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

***simm***

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.71 LDURSH

Load Register Signed Halfword (unscaled).

### Syntax

LDURSH *Wt*, [*Xn/SP{, #simm}*] ; 32-bit

LDURSH *Xt*, [*Xn/SP{, #simm}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.72 LDURSW

Load Register Signed Word (unscaled).

### Syntax

LDURSW *Xt*, [*Xn/SP{, #simm}*]

Where:

**Xt**

Is the 64-bit name of the general-purpose register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**simm**

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.73 LDXP

Load Exclusive Pair of Registers.

### Syntax

LDXP *Wt1*, *Wt2*, [*Xn/SP{, #0}*] ; 32-bit

LDXP *Xt1*, *Xt2*, [*Xn/SP{, #0}*] ; 64-bit

Where:

***Wt1***

Is the 32-bit name of the first general-purpose register to be transferred.

***Wt2***

Is the 32-bit name of the second general-purpose register to be transferred.

***Xt1***

Is the 64-bit name of the first general-purpose register to be transferred.

***Xt2***

Is the 64-bit name of the second general-purpose register to be transferred.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#). For information about memory accesses see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

### Note

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDXP.

---

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.74 LDXR

Load Exclusive Register.

### Syntax

`LDXR Wt, [Xn/SP{,#0}] ; 32-bit`

`LDXR Xt, [Xn/SP{,#0}] ; 64-bit`

Where:

**Wt**

Is the 32-bit name of the general-purpose register to be transferred.

**Xt**

Is the 64-bit name of the general-purpose register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.75 LDXRB

Load Exclusive Register Byte.

### Syntax

`LDXRB Wt, [Xn/SP{,#0}]`

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.76 LDXRH

Load Exclusive Register Halfword.

### Syntax

`LDXRH Wt, [Xn/SP{,#0}]`

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.77 PRFM (immediate)

Prefetch Memory (immediate).

### Syntax

PRFM (*prfop*|#*imm5*), [*Xn/SP*{, #*pimm*}]

Where:

***prfop***

Is the prefetch operation, defined as *type*<*target*><*policy*>.

*type* is one of:

**PLD**

Prefetch for load.

**PLI**

Preload instructions.

**PST**

Prefetch for store.

<*target*> is one of:

**L1**

Level 1 cache.

**L2**

Level 2 cache.

**L3**

Level 3 cache.

<*policy*> is one of:

**KEEP**

Retained or temporal prefetch, allocated in the cache normally.

**STRM**

Streaming or non-temporal prefetch, for data that is used only once.

***imm5***

Is the prefetch operation encoding as an immediate, in the range 0 to 31.

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.

***pimm***

Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

For information about memory accesses, see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

**Related reference**

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.78 PRFM (literal)

Prefetch Memory (literal).

### Syntax

`PRFM (prfop|#imm5), Label`

Where:

**`prfop`**

Is the prefetch operation, defined as `type<target><policy>`.

`type` is one of:

**PLD**

Prefetch for load.

**PLI**

Preload instructions.

**PST**

Prefetch for store.

`<target>` is one of:

**L1**

Level 1 cache.

**L2**

Level 2 cache.

**L3**

Level 3 cache.

`<policy>` is one of:

**KEEP**

Retained or temporal prefetch, allocated in the cache normally.

**STRM**

Streaming or non-temporal prefetch, for data that is used only once.

**`imm5`**

Is the prefetch operation encoding as an immediate, in the range 0 to 31.

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  $\pm 1\text{MB}$ .

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

For information about memory accesses, see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.79 PRFM (register)

Prefetch Memory (register).

### Syntax

PRFM (*prfop*|#*imm5*), [*Xn/SP*, (*Wm*|*Xm*){|, *extend* {*amount*}|}]

Where:

***prfop***

Is the prefetch operation, defined as *type*<*target*><*policy*>.

*type* is one of:

**PLD**

Prefetch for load.

**PLI**

Preload instructions.

**PST**

Prefetch for store.

<*target*> is one of:

**L1**

Level 1 cache.

**L2**

Level 2 cache.

**L3**

Level 3 cache.

<*policy*> is one of:

**KEEP**

Retained or temporal prefetch, allocated in the cache normally.

**STRM**

Streaming or non-temporal prefetch, for data that is used only once.

***imm5***

Is the prefetch operation encoding as an immediate, in the range 0 to 31.

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.

***Wm***

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

***Xm***

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

***extend***

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of UXTW, LSL, SXTW or 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, and can be either #0 or #3.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

For information about memory accesses, see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

**Related reference**

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

**Related information**

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.80 PRFUM (unscaled offset)

Prefetch Memory (unscaled offset).

### Syntax

PRFUM (*prfop*|#*imm5*), [*Xn/SP*{, #*simm*}]

Where:

***prfop***

Is the prefetch operation, defined as *type*<*target*><*policy*>.

*type* is one of:

**PLD**

Prefetch for load.

**PLI**

Preload instructions.

**PST**

Prefetch for store.

<*target*> is one of:

**L1**

Level 1 cache.

**L2**

Level 2 cache.

**L3**

Level 3 cache.

<*policy*> is one of:

**KEEP**

Retained or temporal prefetch, allocated in the cache normally.

**STRM**

Streaming or non-temporal prefetch, for data that is used only once.

***imm5***

Is the prefetch operation encoding as an immediate, in the range 0 to 31.

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.

***simm***

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

For information about memory accesses, see *Load/Store addressing modes* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.81 STADD, STADDL, STADDL

Atomic add on word or doubleword in memory, without return.

### Syntax

STADD *Ws*, [Xn/SP] ; 32-bit, no memory ordering

STADDL *Ws*, [Xn/SP] ; 32-bit, release

STADD *Xs*, [Xn/SP] ; 64-bit, no memory ordering

STADDL *Xs*, [Xn/SP] ; 64-bit, release

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.82 STADDB, STADDLB

Atomic add on byte in memory, without return.

### Syntax

STADDB *Ws*, [X*n*/SP] ; No memory ordering general registers

STADDLB *Ws*, [X*n*/SP] ; Release general registers

Where:

*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.

X*n*/SP

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.83 STADDH, STADDLH

Atomic add on halfword in memory, without return.

### Syntax

STADDH *Ws*, [X*n*/SP] ; No memory ordering general registers

STADDLH *Ws*, [X*n*/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.84 STCLR, STCLRL, STCLRL

Atomic bit clear on word or doubleword in memory, without return.

### Syntax

STCLR *Ws*, [Xn/SP] ; 32-bit, no memory ordering

STCLRL *Ws*, [Xn/SP] ; 32-bit, release

STCLR *Xs*, [Xn/SP] ; 64-bit, no memory ordering

STCLRL *Xs*, [Xn/SP] ; 64-bit, release

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.85 STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return.

### Syntax

STCLRB *Ws*, [Xn/SP] ; No memory ordering general registers

STCLRLB *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.86 STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return.

### Syntax

STCLRH *Ws*, [X*n*/SP] ; No memory ordering general registers

STCLRLH *Ws*, [X*n*/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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.

- STCLRH has no memory ordering semantics.
- STCLRLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.87 STEOR, STEORL, STEORL

Atomic exclusive OR on word or doubleword in memory, without return.

### Syntax

STEOR *Ws*, [Xn/SP] ; 32-bit, no memory ordering

STEORL *Ws*, [Xn/SP] ; 32-bit, release

STEOR *Xs*, [Xn/SP] ; 64-bit, no memory ordering

STEORL *Xs*, [Xn/SP] ; 64-bit, release

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.88 STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return.

### Syntax

STEORB *Ws*, [Xn/SP] ; No memory ordering general registers

STEORLB *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.89 STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return.

### Syntax

STEORH *Ws*, [Xn/SP] ; No memory ordering general registers

STEORLH *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.90 STLLR

Store LORelease Register.

### Syntax

STLLR *Wt*, [*Xn/SP{,#0}*] ; 32-bit

STLLR *Xt*, [*Xn/SP{,#0}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.91 STLLRB

Store Lorelease Register Byte.

### Syntax

STLLRB *Wt*, [*Xn/SP{,#0}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.92 STLLRH

Store LORelease Register Halfword.

### Syntax

STLLRH *Wt*, [*Xn/SP{,#0}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.93 STLR

Store-Release Register.

### Syntax

STLR *Wt*, [*Xn/SP{,#0}*] ; 32-bit

STLR *Xt*, [*Xn/SP{,#0}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.94 STLRB

Store-Release Register Byte.

### Syntax

STLRB *Wt*, [*Xn/SP{, #0}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.95 STLRH

Store-Release Register Halfword.

### Syntax

STLRH *Wt*, [*Xn/SP{, #0}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.96 STLXP

Store-Release Exclusive Pair of registers.

### Syntax

STLXP *Ws*, *Wt1*, *Wt2*, [*Xn/SP{,#0}*] ; 32-bit

STLXP *Ws*, *Xt1*, *Xt2*, [*Xn/SP{,#0}*] ; 64-bit

Where:

*Wt1*

Is the 32-bit name of the first general-purpose register to be transferred.

*Wt2*

Is the 32-bit name of the second general-purpose register to be transferred.

*Xt1*

Is the 64-bit name of the first general-purpose register to be transferred.

*Xt2*

Is the 64-bit name of the second general-purpose register to be transferred.

*Ws*

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is:

**0**

If the operation updates memory.

**1**

If the operation fails to update memory.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### 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:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

———— **Note** ————

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly STLXP.

**Related reference**

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.97 STLXR

Store-Release Exclusive Register.

### Syntax

`STLXR Ws, Wt, [Xn/SP{,#0}] ; 32-bit`

`STLXR Ws, Xt, [Xn/SP{,#0}] ; 64-bit`

Where:

`Wt`

Is the 32-bit name of the general-purpose register to be transferred.

`Xt`

Is the 64-bit name of the general-purpose register to be transferred.

`Ws`

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is:

`0`

If the operation updates memory.

`1`

If the operation fails to update memory.

`Xn/SP`

Is the 64-bit name of the general-purpose base register or stack pointer.

### 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:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#). 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

#### Note

---

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly STLXR.

---

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

***Related information***

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.98 STLXRB

Store-Release Exclusive Register Byte.

### Syntax

STLXRB *Ws*, *Wt*, [*Xn/SP{, #0}*]

Where:

*Ws*

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. 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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

#### Note

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly STLXRB.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.99 STLXRH

Store-Release Exclusive Register Halfword.

### Syntax

STLXRH *Ws*, *Wt*, [*Xn/SP{, #0}*]

Where:

*Ws*

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. 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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### 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:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly STLXRH.

---

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.100 STNP

Store Pair of Registers, with non-temporal hint.

### Syntax

STNP *Wt1*, *Wt2*, [*Xn/SP{, #imm}*] ; 32-bit

STNP *Xt1*, *Xt2*, [*Xn/SP{, #imm}*] ; 64-bit

Where:

*Wt1*

Is the 32-bit name of the first general-purpose register to be transferred.

*Wt2*

Is the 32-bit name of the second general-purpose register to be transferred.

*imm*

Depends on the instruction variant:

#### 32-bit general registers

Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

#### 64-bit general registers

Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0.

*Xt1*

Is the 64-bit name of the first general-purpose register to be transferred.

*Xt2*

Is the 64-bit name of the second general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#). For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.101 STP

Store Pair of Registers.

### Syntax

```
STP Wt1, Wt2, [Xn/SP], #imm ; 32-bit  
STP Xt1, Xt2, [Xn/SP], #imm ; 64-bit  
STP Wt1, Wt2, [Xn/SP, #imm]! ; 32-bit  
STP Xt1, Xt2, [Xn/SP, #imm]! ; 64-bit  
STP Wt1, Wt2, [Xn/SP{, #imm}] ; 32-bit  
STP Xt1, Xt2, [Xn/SP{, #imm}] ; 64-bit
```

Where:

*Wt1*

Is the 32-bit name of the first general-purpose register to be transferred.

*Wt2*

Is the 32-bit name of the second general-purpose register to be transferred.

*imm*

Depends on the instruction variant:

#### 32-bit general registers

Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

#### 64-bit general registers

Is the signed immediate byte offset, a multiple of 8 in the range -512 to 504.

*Xt1*

Is the 64-bit name of the first general-purpose register to be transferred.

*Xt2*

Is the 64-bit name of the second general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

#### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly STP.

---

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.102 STR (immediate)

Store Register (immediate).

### Syntax

```
STR Wt, [Xn/SP], #simm ; 32-bit  
STR Xt, [Xn/SP], #simm ; 64-bit  
STR Wt, [Xn/SP, #simm]! ; 32-bit  
STR Xt, [Xn/SP, #simm]! ; 64-bit  
STR Wt, [Xn/SP{, #pimm}] ; 32-bit  
STR Xt, [Xn/SP{, #pimm}] ; 64-bit
```

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*simm*

Is the signed immediate byte offset, in the range -256 to 255.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*pimm*

Depends on the instruction variant:

#### 32-bit general registers

Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

#### 64-bit general registers

Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.103 STR (register)

Store Register (register).

### Syntax

STR *Wt*, [*Xn/SP*, (*Wm|Xm*){}*, extend {amount}*{}]] ; 32-bit

STR *Xt*, [*Xn/SP*, (*Wm|Xm*){}*, extend {amount}*{}]] ; 64-bit

Where:

***Wt***

Is the 32-bit name of the general-purpose register to be transferred.

***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:

**32-bit general registers**

Can be one of #0 or #2.

**64-bit general registers**

Can be one of #0 or #3.

***Xt***

Is the 64-bit name of the general-purpose register to be transferred.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

***Wm***

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

***Xm***

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

***extend***

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of the values shown in Usage.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.104 STRB (immediate)

Store Register Byte (immediate).

### Syntax

```
STRB Wt, [Xn/SP], #simm ; Post-index general registers  
STRB Wt, [Xn/SP, #simm]! ; Pre-index general registers  
STRB Wt, [Xn/SP{, #pimm}] ; Unsigned offset general registers
```

Where:

*simm*

Is the signed immediate byte offset, in the range -256 to 255.

*pimm*

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly STRB (immediate).

---

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.105 STRB (register)

Store Register Byte (register).

### Syntax

STRB *Wt*, [Xn/SP, (*Wm|Xm*), *extend {amount}*] ; Extended register general registers

STRB *Wt*, [Xn/SP, *Xm{, LSL amount}*] ; Shifted register general registers

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Wm*

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

*Xm*

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

*extend*

Is the index extend specifier, and can be one of the values shown in Usage.

*amount*

Is the index shift amount, it must be.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.106 STRH (immediate)

Store Register Halfword (immediate).

### Syntax

```
STRH Wt, [Xn/SP], #simm ; Post-index general registers  
STRH Wt, [Xn/SP, #simm]! ; Pre-index general registers  
STRH Wt, [Xn/SP{, #pimm}] ; Unsigned offset general registers
```

Where:

*simm*

Is the signed immediate byte offset, in the range -256 to 255.

*pimm*

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Note

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly STRH (immediate).

---

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.107 STRH (register)

Store Register Halfword (register).

### Syntax

STRH *Wt*, [X*n*/SP, (*Wm*|*Xm*){}, *extend* {*amount*}]]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Wm*

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

*Xm*

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

*extend*

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of UXTW, LSL, SXTW or 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, and can be either #0 or #1.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.108 STSET, STSETL, STSETL

Atomic bit set on word or doubleword in memory, without return.

### Syntax

STSET *Ws*, [Xn/SP] ; 32-bit, no memory ordering

STSETL *Ws*, [Xn/SP] ; 32-bit, release

STSET *Xs*, [Xn/SP] ; 64-bit, no memory ordering

STSETL *Xs*, [Xn/SP] ; 64-bit, release

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.109 STSETB, STSETLB

Atomic bit set on byte in memory, without return.

### Syntax

STSETB *Ws*, [Xn/SP] ; No memory ordering general registers

STSETLB *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.110 STSETH, STSETLH

Atomic bit set on halfword in memory, without return.

### Syntax

STSETH *Ws*, [X*n*/SP] ; No memory ordering general registers

STSETLH *Ws*, [X*n*/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D3.111 STSMAX, STSMAXL, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return.

### Syntax

STS MAX *Ws*, [X*n*/SP] ; 32-bit, no memory ordering

STS MAXL *Ws*, [X*n*/SP] ; 32-bit, release

STS MAX *Xs*, [X*n*/SP] ; 64-bit, no memory ordering

STS MAXL *Xs*, [X*n*/SP] ; 64-bit, release

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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.

- STS MAX has no memory ordering semantics.
- STS MAXL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.112 STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return.

### Syntax

STSMAXB *Ws*, [Xn/SP] ; No memory ordering general registers

STSMAXLB *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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.
- STSMAXLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.113 STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return.

### Syntax

STSMAXH *Ws*, [Xn/SP] ; No memory ordering general registers

STSMAXLH *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.114 STSMIN, STSMINL, STSMINL

Atomic signed minimum on word or doubleword in memory, without return.

### Syntax

STSMIN *Ws*, [Xn/SP] ; 32-bit, no memory ordering

STSMINL *Ws*, [Xn/SP] ; 32-bit, release

STSMIN *Xs*, [Xn/SP] ; 64-bit, no memory ordering

STSMINL *Xs*, [Xn/SP] ; 64-bit, release

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.115 STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return.

### Syntax

STSMINB *Ws*, [Xn/SP] ; No memory ordering general registers

STSMINLB *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.116 STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return.

### Syntax

STSMINH *Ws*, [Xn/SP] ; No memory ordering general registers

STSMINLH *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.117 STTR

Store Register (unprivileged).

### Syntax

STTR *Wt*, [*Xn/SP{, #simm}*] ; 32-bit

STTR *Xt*, [*Xn/SP{, #simm}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.118 STTRB

Store Register Byte (unprivileged).

### Syntax

STTRB *Wt*, [*Xn/SP{, #simm}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.119 STTRH

Store Register Halfword (unprivileged).

### Syntax

STTRH *Wt*, [*Xn/SP{, #simm}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.120 STUMAX, STUMAXL, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return.

### Syntax

STUMAX *Ws*, [Xn/SP] ; 32-bit, no memory ordering

STUMAXL *Ws*, [Xn/SP] ; 32-bit, release

STUMAX *Xs*, [Xn/SP] ; 64-bit, no memory ordering

STUMAXL *Xs*, [Xn/SP] ; 64-bit, release

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.121 STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return.

### Syntax

STUMAXB *Ws*, [Xn/SP] ; No memory ordering general registers

STUMAXLB *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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.
- STUMAXLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.122 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return.

### Syntax

STUMAXH *Ws*, [Xn/SP] ; No memory ordering general registers

STUMAXLH *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.123 STUMIN, STUMINL, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return.

### Syntax

STUMIN *Ws*, [Xn/SP] ; 32-bit, no memory ordering

STUMINL *Ws*, [Xn/SP] ; 32-bit, release

STUMIN *Xs*, [Xn/SP] ; 64-bit, no memory ordering

STUMINL *Xs*, [Xn/SP] ; 64-bit, release

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*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.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.124 STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return.

### Syntax

STUMINB *Ws*, [Xn/SP] ; No memory ordering general registers

STUMINLB *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.125 STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return.

### Syntax

STUMINH *Ws*, [Xn/SP] ; No memory ordering general registers

STUMINLH *Ws*, [Xn/SP] ; Release general registers

Where:

*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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.126 STUR

Store Register (unscaled).

### Syntax

STUR *Wt*, [*Xn/SP{*, #*simm**}]* ; 32-bit

STUR *Xt*, [*Xn/SP{*, #*simm**}]* ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.127 STURB

Store Register Byte (unscaled).

### Syntax

STURB *Wt*, [*Xn/SP{*, *#simm**}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.128 STURH

Store Register Halfword (unscaled).

### Syntax

STURH *Wt*, [*Xn/SP{, #simm}*]

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*simm*

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.129 STXP

Store Exclusive Pair of registers.

### Syntax

STXP *Ws*, *Wt1*, *Wt2*, [*Xn/SP{,#0}*] ; 32-bit

STXP *Ws*, *Xt1*, *Xt2*, [*Xn/SP{,#0}*] ; 64-bit

Where:

***Wt1***

Is the 32-bit name of the first general-purpose register to be transferred.

***Wt2***

Is the 32-bit name of the second general-purpose register to be transferred.

***Xt1***

Is the 64-bit name of the first general-purpose register to be transferred.

***Xt2***

Is the 64-bit name of the second general-purpose register to be transferred.

***Ws***

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is:

**0**

If the operation updates memory.

**1**

If the operation fails to update memory.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

### 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:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

— Note —

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly STXP.

**Related reference**

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.130 STXR

Store Exclusive Register.

### Syntax

STXR *Ws*, *Wt*, [*Xn/SP{,#0}*] ; 32-bit

STXR *Ws*, *Xt*, [*Xn/SP{,#0}*] ; 64-bit

Where:

*Wt*

Is the 32-bit name of the general-purpose register to be transferred.

*Xt*

Is the 64-bit name of the general-purpose register to be transferred.

*Ws*

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is:

**0**

If the operation updates memory.

**1**

If the operation fails to update memory.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### 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:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly STXR.

---

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

***Related information***

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.131 STXRB

Store Exclusive Register Byte.

### Syntax

STXRB *Ws*, *Wt*, [*Xn/SP{, #0}*]

Where:

*Ws*

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. 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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

#### Note

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, and particularly STXRB.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.132 STXRH

Store Exclusive Register Halfword.

### Syntax

STXRH *Ws*, *Wt*, [*Xn/SP{, #0}*]

Where:

*Ws*

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. 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.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### 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:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.133 SWPA, SWPAL, SWP, SWPL, SWPAL, SWP, SWPL

Swap word or doubleword in memory.

### Syntax

```
SWPA Ws, Wt, [Xn/SP] ; 32-bit, acquire  
SWPAL Ws, Wt, [Xn/SP] ; 32-bit, acquire and release  
SWP Ws, Wt, [Xn/SP] ; 32-bit, no memory ordering  
SWPL Ws, Wt, [Xn/SP] ; 32-bit, release  
SWPA Xs, Xt, [Xn/SP] ; 64-bit, acquire  
SWPAL Xs, Xt, [Xn/SP] ; 64-bit, acquire and release  
SWP Xs, Xt, [Xn/SP] ; 64-bit, no memory ordering  
SWPL Xs, Xt, [Xn/SP] ; 64-bit, release
```

Where:

*Ws*

Is the 32-bit name of the general-purpose register to be stored.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

*Xs*

Is the 64-bit name of the general-purpose register to be stored.

*Xt*

Is the 64-bit name of the general-purpose register to be loaded.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.134 SWPAB, SWPALB, SWPB, SWPLB

Swap byte in memory.

### Syntax

```
SWPAB Ws, Wt, [Xn/SP] ; Acquire general registers  
SWPALB Ws, Wt, [Xn/SP] ; Acquire and release general registers  
SWPB Ws, Wt, [Xn/SP] ; No memory ordering general registers  
SWPLB Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*Ws*

Is the 32-bit name of the general-purpose register to be stored.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D3.135 SWPAH, SWPALH, SWPH, SWPLH

Swap halfword in memory.

### Syntax

```
SWPAH Ws, Wt, [Xn/SP] ; Acquire general registers  
SWPALH Ws, Wt, [Xn/SP] ; Acquire and release general registers  
SWPH Ws, Wt, [Xn/SP] ; No memory ordering general registers  
SWPLH Ws, Wt, [Xn/SP] ; Release general registers
```

Where:

*Ws*

Is the 32-bit name of the general-purpose register to be stored.

*Wt*

Is the 32-bit name of the general-purpose register to be loaded.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Architectures supported

Supported in the Armv8.1 architecture and later.

### Usage

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Related reference

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*



# Chapter D4

## A64 Floating-point Instructions

Describes the A64 floating-point instructions.

It contains the following sections:

- [D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029.
- [D4.2 Register restrictions for A64 instructions](#) on page D4-1032.
- [D4.3 FABS \(scalar\)](#) on page D4-1033.
- [D4.4 FADD \(scalar\)](#) on page D4-1034.
- [D4.5 FCCMP](#) on page D4-1035.
- [D4.6 FCCMPE](#) on page D4-1036.
- [D4.7 FCMP](#) on page D4-1038.
- [D4.8 FCMPE](#) on page D4-1040.
- [D4.9 FCSEL](#) on page D4-1042.
- [D4.10 FCVT](#) on page D4-1043.
- [D4.11 FCVTAU \(scalar\)](#) on page D4-1044.
- [D4.12 FCVTAS \(scalar\)](#) on page D4-1045.
- [D4.13 FCVTAU \(scalar\)](#) on page D4-1046.
- [D4.14 FCVTPS \(scalar\)](#) on page D4-1047.
- [D4.15 FCVTPU \(scalar\)](#) on page D4-1048.
- [D4.16 FCVTPS \(scalar\)](#) on page D4-1049.
- [D4.17 FCVTPU \(scalar\)](#) on page D4-1050.
- [D4.18 FCVTPS \(scalar\)](#) on page D4-1051.
- [D4.19 FCVTPU \(scalar, fixed-point\)](#) on page D4-1052.
- [D4.20 FCVTPS \(scalar, integer\)](#) on page D4-1054.
- [D4.21 FCVTPU \(scalar, fixed-point\)](#) on page D4-1055.
- [D4.22 FCVTPU \(scalar, integer\)](#) on page D4-1057.
- [D4.23 FDIV \(scalar\)](#) on page D4-1058.

- [D4.24 FJCVTZS](#) on page D4-1059.
- [D4.25 FMADD](#) on page D4-1060.
- [D4.26 FMAX \(scalar\)](#) on page D4-1061.
- [D4.27 FMAXNM \(scalar\)](#) on page D4-1062.
- [D4.28 FMIN \(scalar\)](#) on page D4-1063.
- [D4.29 FMINNM \(scalar\)](#) on page D4-1064.
- [D4.30 FMOV \(register\)](#) on page D4-1065.
- [D4.31 FMOV \(general\)](#) on page D4-1066.
- [D4.32 FMOV \(scalar, immediate\)](#) on page D4-1067.
- [D4.33 FMSUB](#) on page D4-1068.
- [D4.34 FMUL \(scalar\)](#) on page D4-1069.
- [D4.35 FNEG \(scalar\)](#) on page D4-1070.
- [D4.36 FNMMADD](#) on page D4-1071.
- [D4.37 FNMSUB](#) on page D4-1072.
- [D4.38 FNMUL \(scalar\)](#) on page D4-1073.
- [D4.39 FRINTA \(scalar\)](#) on page D4-1074.
- [D4.40 FRINTI \(scalar\)](#) on page D4-1075.
- [D4.41 FRINTM \(scalar\)](#) on page D4-1076.
- [D4.42 FRINTN \(scalar\)](#) on page D4-1077.
- [D4.43 FRINTP \(scalar\)](#) on page D4-1078.
- [D4.44 FRINTX \(scalar\)](#) on page D4-1079.
- [D4.45 FRINTZ \(scalar\)](#) on page D4-1080.
- [D4.46 FSQRT \(scalar\)](#) on page D4-1081.
- [D4.47 FSUB \(scalar\)](#) on page D4-1082.
- [D4.48 LDNP \(SIMD and FP\)](#) on page D4-1083.
- [D4.49 LDP \(SIMD and FP\)](#) on page D4-1085.
- [D4.50 LDR \(immediate, SIMD and FP\)](#) on page D4-1087.
- [D4.51 LDR \(literal, SIMD and FP\)](#) on page D4-1089.
- [D4.52 LDR \(register, SIMD and FP\)](#) on page D4-1090.
- [D4.53 LDUR \(SIMD and FP\)](#) on page D4-1092.
- [D4.54 SCVTF \(scalar, fixed-point\)](#) on page D4-1093.
- [D4.55 SCVTF \(scalar, integer\)](#) on page D4-1095.
- [D4.56 STNP \(SIMD and FP\)](#) on page D4-1096.
- [D4.57 STP \(SIMD and FP\)](#) on page D4-1097.
- [D4.58 STR \(immediate, SIMD and FP\)](#) on page D4-1098.
- [D4.59 STR \(register, SIMD and FP\)](#) on page D4-1100.
- [D4.60 STUR \(SIMD and FP\)](#) on page D4-1102.
- [D4.61 UCVTF \(scalar, fixed-point\)](#) on page D4-1103.
- [D4.62 UCVTF \(scalar, integer\)](#) on page D4-1105.

## D4.1 A64 floating-point instructions in alphabetical order

A summary of the A64 floating-point instructions that are supported.

**Table D4-1 Summary of A64 floating-point instructions**

Mnemonic	Brief description	See
FABS (scalar)	Floating-point Absolute value (scalar)	<a href="#">D4.3 FABS (scalar) on page D4-1033</a>
FADD (scalar)	Floating-point Add (scalar)	<a href="#">D4.4 FADD (scalar) on page D4-1034</a>
FCCMP	Floating-point Conditional quiet Compare (scalar)	<a href="#">D4.5 FCCMP on page D4-1035</a>
FCCMPE	Floating-point Conditional signaling Compare (scalar)	<a href="#">D4.6 FCCMPE on page D4-1036</a>
FCMP	Floating-point quiet Compare (scalar)	<a href="#">D4.7 FCMP on page D4-1038</a>
FCMPE	Floating-point signaling Compare (scalar)	<a href="#">D4.8 FCMPE on page D4-1040</a>
FCSEL	Floating-point Conditional Select (scalar)	<a href="#">D4.9 FCSEL on page D4-1042</a>
FCVT	Floating-point Convert precision (scalar)	<a href="#">D4.10 FCVT on page D4-1043</a>
FCVTAS (scalar)	Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar)	<a href="#">D4.11 FCVTAS (scalar) on page D4-1044</a>
FCVTAU (scalar)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar)	<a href="#">D4.12 FCVTAU (scalar) on page D4-1045</a>
FCVTMS (scalar)	Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar)	<a href="#">D4.13 FCVTMS (scalar) on page D4-1046</a>
FCVTMU (scalar)	Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar)	<a href="#">D4.14 FCVTMU (scalar) on page D4-1047</a>
FCVTNS (scalar)	Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar)	<a href="#">D4.15 FCVTNS (scalar) on page D4-1048</a>
FCVTNU (scalar)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar)	<a href="#">D4.16 FCVTNU (scalar) on page D4-1049</a>
FCVTPS (scalar)	Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar)	<a href="#">D4.17 FCVTPS (scalar) on page D4-1050</a>
FCVTPU (scalar)	Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar)	<a href="#">D4.18 FCVTPU (scalar) on page D4-1051</a>
FCVTZS (scalar, fixed-point)	Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar)	<a href="#">D4.19 FCVTZS (scalar, fixed-point) on page D4-1052</a>
FCVTZS (scalar, integer)	Floating-point Convert to Signed integer, rounding toward Zero (scalar)	<a href="#">D4.20 FCVTZS (scalar, integer) on page D4-1054</a>
FCVTZU (scalar, fixed-point)	Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar)	<a href="#">D4.21 FCVTZU (scalar; fixed-point) on page D4-1055</a>
FCVTZU (scalar, integer)	Floating-point Convert to Unsigned integer, rounding toward Zero (scalar)	<a href="#">D4.22 FCVTZU (scalar; integer) on page D4-1057</a>
FDIV (scalar)	Floating-point Divide (scalar)	<a href="#">D4.23 FDIV (scalar) on page D4-1058</a>
FJCVTZS	Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero	<a href="#">D4.24 FJCVTZS on page D4-1059</a>

**Table D4-1 Summary of A64 floating-point instructions (continued)**

Mnemonic	Brief description	See
FMADD	Floating-point fused Multiply-Add (scalar)	<a href="#">D4.25 FMADD on page D4-1060</a>
FMAX (scalar)	Floating-point Maximum (scalar)	<a href="#">D4.26 FMAX (scalar) on page D4-1061</a>
FMAXNM (scalar)	Floating-point Maximum Number (scalar)	<a href="#">D4.27 FMAXNM (scalar) on page D4-1062</a>
FMIN (scalar)	Floating-point Minimum (scalar)	<a href="#">D4.28 FMIN (scalar) on page D4-1063</a>
FMINNM (scalar)	Floating-point Minimum Number (scalar)	<a href="#">D4.29 FMINNM (scalar) on page D4-1064</a>
FMOV (register)	Floating-point Move register without conversion	<a href="#">D4.30 FMOV (register) on page D4-1065</a>
FMOV (general)	Floating-point Move to or from general-purpose register without conversion	<a href="#">D4.31 FMOV (general) on page D4-1066</a>
FMOV (scalar, immediate)	Floating-point move immediate (scalar)	<a href="#">D4.32 FMOV (scalar, immediate) on page D4-1067</a>
FMSUB	Floating-point Fused Multiply-Subtract (scalar)	<a href="#">D4.33 FMSUB on page D4-1068</a>
FMUL (scalar)	Floating-point Multiply (scalar)	<a href="#">D4.34 FMUL (scalar) on page D4-1069</a>
FNEG (scalar)	Floating-point Negate (scalar)	<a href="#">D4.35 FNEG (scalar) on page D4-1070</a>
FNMADD	Floating-point Negated fused Multiply-Add (scalar)	<a href="#">D4.36 FNMADD on page D4-1071</a>
FNMSUB	Floating-point Negated fused Multiply-Subtract (scalar)	<a href="#">D4.37 FNMSUB on page D4-1072</a>
FNMUL (scalar)	Floating-point Multiply-Negate (scalar)	<a href="#">D4.38 FNMUL (scalar) on page D4-1073</a>
FRINTA (scalar)	Floating-point Round to Integral, to nearest with ties to Away (scalar)	<a href="#">D4.39 FRINTA (scalar) on page D4-1074</a>
FRINTI (scalar)	Floating-point Round to Integral, using current rounding mode (scalar)	<a href="#">D4.40 FRINTI (scalar) on page D4-1075</a>
FRINTM (scalar)	Floating-point Round to Integral, toward Minus infinity (scalar)	<a href="#">D4.41 FRINTM (scalar) on page D4-1076</a>
FRINTN (scalar)	Floating-point Round to Integral, to nearest with ties to even (scalar)	<a href="#">D4.42 FRINTN (scalar) on page D4-1077</a>
FRINTP (scalar)	Floating-point Round to Integral, toward Plus infinity (scalar)	<a href="#">D4.43 FRINTP (scalar) on page D4-1078</a>
FRINTX (scalar)	Floating-point Round to Integral exact, using current rounding mode (scalar)	<a href="#">D4.44 FRINTX (scalar) on page D4-1079</a>
FRINTZ (scalar)	Floating-point Round to Integral, toward Zero (scalar)	<a href="#">D4.45 FRINTZ (scalar) on page D4-1080</a>
FSQRT (scalar)	Floating-point Square Root (scalar)	<a href="#">D4.46 FSQRT (scalar) on page D4-1081</a>
FSUB (scalar)	Floating-point Subtract (scalar)	<a href="#">D4.47 FSUB (scalar) on page D4-1082</a>
LDNP (SIMD and FP)	Load Pair of SIMD and FP registers, with Non-temporal hint	<a href="#">D4.48 LDNP (SIMD and FP) on page D4-1083</a>
LDP (SIMD and FP)	Load Pair of SIMD and FP registers	<a href="#">D4.49 LDP (SIMD and FP) on page D4-1085</a>
LDR (immediate, SIMD and FP)	Load SIMD and FP Register (immediate offset)	<a href="#">D4.50 LDR (immediate, SIMD and FP) on page D4-1087</a>

**Table D4-1 Summary of A64 floating-point instructions (continued)**

Mnemonic	Brief description	See
LDR (literal, SIMD and FP)	Load SIMD and FP Register (PC-relative literal)	<a href="#">D4.51 LDR (literal, SIMD and FP) on page D4-1089</a>
LDR (register, SIMD and FP)	Load SIMD and FP Register (register offset)	<a href="#">D4.52 LDR (register, SIMD and FP) on page D4-1090</a>
LDUR (SIMD and FP)	Load SIMD and FP Register (unscaled offset)	<a href="#">D4.53 LDUR (SIMD and FP) on page D4-1092</a>
SCVTF (scalar, fixed-point)	Signed fixed-point Convert to Floating-point (scalar)	<a href="#">D4.54 SCVTF (scalar, fixed-point) on page D4-1093</a>
SCVTF (scalar, integer)	Signed integer Convert to Floating-point (scalar)	<a href="#">D4.55 SCVTF (scalar, integer) on page D4-1095</a>
STNP (SIMD and FP)	Store Pair of SIMD and FP registers, with Non-temporal hint	<a href="#">D4.56 STNP (SIMD and FP) on page D4-1096</a>
STP (SIMD and FP)	Store Pair of SIMD and FP registers	<a href="#">D4.57 STP (SIMD and FP) on page D4-1097</a>
STR (immediate, SIMD and FP)	Store SIMD and FP register (immediate offset)	<a href="#">D4.58 STR (immediate, SIMD and FP) on page D4-1098</a>
STR (register, SIMD and FP)	Store SIMD and FP register (register offset)	<a href="#">D4.59 STR (register, SIMD and FP) on page D4-1100</a>
STUR (SIMD and FP)	Store SIMD and FP register (unscaled offset)	<a href="#">D4.60 STUR (SIMD and FP) on page D4-1102</a>
UCVTF (scalar, fixed-point)	Unsigned fixed-point Convert to Floating-point (scalar)	<a href="#">D4.61 UCVTF (scalar, fixed-point) on page D4-1103</a>
UCVTF (scalar, integer)	Unsigned integer Convert to Floating-point (scalar)	<a href="#">D4.62 UCVTF (scalar, integer) on page D4-1105</a>

## D4.2 Register restrictions for A64 instructions

In A64 instructions, the general-purpose integer registers are W0-W30 for 32-bit registers and X0-X30 for 64-bit registers.

You cannot refer to register 31 by number. In a few instructions, you can refer to it using one of the following names:

**WSP**

the current stack pointer in a 32-bit context.

**SP**

the current stack pointer in a 64-bit context.

**WZR**

the zero register in a 32-bit context.

**XZR**

the zero register in a 64-bit context.

You can only use one of these names if it is mentioned in the Syntax section for the instruction.

You cannot refer to the Program Counter (PC) explicitly by name or by number.

## D4.3 FABS (scalar)

Floating-point Absolute value (scalar).

### Syntax

```
FABS Hd, Hn ; Half-precision  
FABS Sd, Sn ; Single-precision  
FABS Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD and FP source register and writes the result to the SIMD and 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.

*Vd* = *abs(Vn)*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

## D4.4 FADD (scalar)

Floating-point Add (scalar).

### Syntax

FADD *Hd*, *Hn*, *Hm* ; Half-precision

FADD *Sd*, *Sn*, *Sm* ; Single-precision

FADD *Dd*, *Dn*, *Dm* ; Double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD and FP registers, and writes the result to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

$Vd = Vn +Vm$ .

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.5 FCCMP

Floating-point Conditional quiet Compare (scalar).

### Syntax

```
FCCMP Hn, Hm, #nzcv, cond ; Half-precision
FCCMP Sn, Sm, #nzcv, cond ; Single-precision
FCCMP Dn, Dm, #nzcv, cond ; Double-precision
```

Where:

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

*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.

*cond*

Is one of the standard conditions.

### 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

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

```
flags = if cond then compareQuiet(Vn,Vm) else #nzcv.
```

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.6 FCCMPE

Floating-point Conditional signaling Compare (scalar).

### Syntax

```
FCCMPE Hn, Hm, #nzcv, cond ; Half-precision
FCCMPE Sn, Sm, #nzcv, cond ; Single-precision
FCCMPE Dn, Dm, #nzcv, cond ; Double-precision
```

Where:

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

*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.

*cond*

Is one of the standard conditions.

### 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

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

```
flags = if cond then compareSignaling(Vn,Vm) else #nzcv.
```

**Related reference**

*D1.8 Condition code suffixes and related flags* on page D1-655

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.7 FCMP

Floating-point quiet Compare (scalar).

### Syntax

```
FCMP Hn, Hm ; Half-precision  
FCMP Hn, #0.0 ; Half-precision, zero  
FCMP Sn, Sm ; Single-precision  
FCMP Sn, #0.0 ; Single-precision, zero  
FCMP Dn, Dm ; Double-precision  
FCMP Dn, #0.0 ; Double-precision, zero
```

Where:

*Hn*

Depends on the instruction variant:

#### Half-precision

Is the 16-bit name of the first SIMD and FP source register

#### Half-precision, zero

Is the 16-bit name of the SIMD and FP source register

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sn*

Depends on the instruction variant:

#### Single-precision

Is the 32-bit name of the first SIMD and FP source register.

#### Single-precision, zero

Is the 32-bit name of the SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dn*

Depends on the instruction variant:

#### Double-precision

Is the 64-bit name of the first SIMD and FP source register.

#### Double-precision, zero

Is the 64-bit name of the SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### 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.

## Usage

Floating-point quiet Compare (scalar). This instruction compares the two SIMD and FP source register values, or the first SIMD and FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.8 FCMPE

Floating-point signaling Compare (scalar).

### Syntax

```
FCMPE Hn, Hm ; Half-precision
FCMPE Hn, #0.0 ; Half-precision, zero
FCMPE Sn, Sm ; Single-precision
FCMPE Sn, #0.0 ; Single-precision, zero
FCMPE Dn, Dm ; Double-precision
FCMPE Dn, #0.0 ; Double-precision, zero
```

Where:

*Hn*

Depends on the instruction variant:

#### Half-precision

Is the 16-bit name of the first SIMD and FP source register

#### Half-precision, zero

Is the 16-bit name of the SIMD and FP source register

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sn*

Depends on the instruction variant:

#### Single-precision

Is the 32-bit name of the first SIMD and FP source register.

#### Single-precision, zero

Is the 32-bit name of the SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dn*

Depends on the instruction variant:

#### Double-precision

Is the 64-bit name of the first SIMD and FP source register.

#### Double-precision, zero

Is the 64-bit name of the SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### 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.

FCMPE 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.

## Usage

Floating-point signaling Compare (scalar). This instruction compares the two SIMD and FP source register values, or the first SIMD and FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.9 FCSEL

Floating-point Conditional Select (scalar).

### Syntax

```
FCSEL Hd, Hn, Hm, cond ; Half-precision  
FCSEL Sd, Sn, Sm, cond ; Single-precision  
FCSEL Dd, Dn, Dm, cond ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

*cond*

Is one of the standard conditions.

### Operation

Floating-point Conditional Select (scalar). This instruction allows the SIMD and FP destination register to take the value from either one or the other of two SIMD and FP source registers. If the condition passes, the first SIMD and FP source register value is taken, otherwise the second SIMD and FP source register value is taken.

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.

*Vd* = if *cond* then *Vn* else *Vm*.

### Related reference

[D1.8 Condition code suffixes and related flags](#) on page D1-655

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

## D4.10 FCVT

Floating-point Convert precision (scalar).

### Syntax

```
FCVT Sd, Hn ; Half-precision to single-precision  
FCVT Dd, Hn ; Half-precision to double-precision  
FCVT Hd, Sn ; Single-precision to half-precision  
FCVT Dd, Sn ; Single-precision to double-precision  
FCVT Hd, Dn ; Double-precision to half-precision  
FCVT Sd, Dn ; Double-precision to single-precision
```

Where:

- Sd* Is the 32-bit name of the SIMD and FP destination register.
- Hn* Is the 16-bit name of the SIMD and FP source register.
- Dd* Is the 64-bit name of the SIMD and FP destination register.
- Hd* Is the 16-bit name of the SIMD and FP destination register.
- Sn* Is the 32-bit name of the SIMD and FP source register.
- Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD and FP source register to the precision for the destination register data type using the rounding mode that is determined by the FPCR and writes the result to the SIMD and 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.

*Vd* = convertFormat(*Vn*), where *V* is D, H, or S.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

## D4.11 FCVTAS (scalar)

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

### Syntax

```
FCVTAS Wd, Hn ; Half-precision to 32-bit  
FCVTAS Xd, Hn ; Half-precision to 64-bit  
FCVTAS Wd, Sn ; Single-precision to 32-bit  
FCVTAS Xd, Sn ; Single-precision to 64-bit  
FCVTAS Wd, Dn ; Double-precision to 32-bit  
FCVTAS Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.
- Hn* Is the 16-bit name of the SIMD and FP source register.
- Xd* Is the 64-bit name of the general-purpose destination register.
- Sn* Is the 32-bit name of the SIMD and FP source register.
- Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `signed_convertToIntegerExactTiesToAway(Vn)`, where *R* is either *W* or *X*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.12 FCVTAU (scalar)

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

### Syntax

```
FCVTAU Wd, Hn ; Half-precision to 32-bit  
FCVTAU Xd, Hn ; Half-precision to 64-bit  
FCVTAU Wd, Sn ; Single-precision to 32-bit  
FCVTAU Xd, Sn ; Single-precision to 64-bit  
FCVTAU Wd, Dn ; Double-precision to 32-bit  
FCVTAU Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.
- Hn* Is the 16-bit name of the SIMD and FP source register.
- Xd* Is the 64-bit name of the general-purpose destination register.
- Sn* Is the 32-bit name of the SIMD and FP source register.
- Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `unsigned_convertToIntegerExactTiesToAway(Vn)`, where *R* is either *W* or *X*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.13 FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).

### Syntax

```
FCVTMS Wd, Hn ; Half-precision to 32-bit  
FCVTMS Xd, Hn ; Half-precision to 64-bit  
FCVTMS Wd, Sn ; Single-precision to 32-bit  
FCVTMS Xd, Sn ; Single-precision to 64-bit  
FCVTMS Wd, Dn ; Double-precision to 32-bit  
FCVTMS Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.
- Hn* Is the 16-bit name of the SIMD and FP source register.
- Xd* Is the 64-bit name of the general-purpose destination register.
- Sn* Is the 32-bit name of the SIMD and FP source register.
- Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `signed_convertToIntegerExactTowardNegative(Vn)`, where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.14 FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).

### Syntax

```
FCVTMU Wd, Hn ; Half-precision to 32-bit  
FCVTMU Xd, Hn ; Half-precision to 64-bit  
FCVTMU Wd, Sn ; Single-precision to 32-bit  
FCVTMU Xd, Sn ; Single-precision to 64-bit  
FCVTMU Wd, Dn ; Double-precision to 32-bit  
FCVTMU Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.  
*Hn* Is the 16-bit name of the SIMD and FP source register.  
*Xd* Is the 64-bit name of the general-purpose destination register.  
*Sn* Is the 32-bit name of the SIMD and FP source register.  
*Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `unsigned_convertToIntegerExactTowardNegative(Vn)`, where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.15 FCVTNS (scalar)

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

### Syntax

```
FCVTNS Wd, Hn ; Half-precision to 32-bit  
FCVTNS Xd, Hn ; Half-precision to 64-bit  
FCVTNS Wd, Sn ; Single-precision to 32-bit  
FCVTNS Xd, Sn ; Single-precision to 64-bit  
FCVTNS Wd, Dn ; Double-precision to 32-bit  
FCVTNS Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.
- Hn* Is the 16-bit name of the SIMD and FP source register.
- Xd* Is the 64-bit name of the general-purpose destination register.
- Sn* Is the 32-bit name of the SIMD and FP source register.
- Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `signed_convertToIntegerExactTiesToEven(Vn)`, where *R* is either *W* or *X*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.16 FCVNU (scalar)

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

### Syntax

```
FCVNU Wd, Hn ; Half-precision to 32-bit  
FCVNU Xd, Hn ; Half-precision to 64-bit  
FCVNU Wd, Sn ; Single-precision to 32-bit  
FCVNU Xd, Sn ; Single-precision to 64-bit  
FCVNU Wd, Dn ; Double-precision to 32-bit  
FCVNU Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.
- Hn* Is the 16-bit name of the SIMD and FP source register.
- Xd* Is the 64-bit name of the general-purpose destination register.
- Sn* Is the 32-bit name of the SIMD and FP source register.
- Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `unsigned_convertToIntegerExactTiesToEven(Vn)`, where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.17 FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).

### Syntax

```
FCVTPS Wd, Hn ; Half-precision to 32-bit  
FCVTPS Xd, Hn ; Half-precision to 64-bit  
FCVTPS Wd, Sn ; Single-precision to 32-bit  
FCVTPS Xd, Sn ; Single-precision to 64-bit  
FCVTPS Wd, Dn ; Double-precision to 32-bit  
FCVTPS Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.  
*Hn* Is the 16-bit name of the SIMD and FP source register.  
*Xd* Is the 64-bit name of the general-purpose destination register.  
*Sn* Is the 32-bit name of the SIMD and FP source register.  
*Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `signed_convertToIntegerExactTowardPositive(Vn)`, where *R* is either *W* or *X*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.18 FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).

### Syntax

```
FCVTPU Wd, Hn ; Half-precision to 32-bit  
FCVTPU Xd, Hn ; Half-precision to 64-bit  
FCVTPU Wd, Sn ; Single-precision to 32-bit  
FCVTPU Xd, Sn ; Single-precision to 64-bit  
FCVTPU Wd, Dn ; Double-precision to 32-bit  
FCVTPU Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.
- Hn* Is the 16-bit name of the SIMD and FP source register.
- Xd* Is the 64-bit name of the general-purpose destination register.
- Sn* Is the 32-bit name of the SIMD and FP source register.
- Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `unsigned_convertToIntegerExactTowardPositive(Vn)`, where *R* is either *W* or *X*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.19 FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).

### Syntax

```
FCVTZS Wd, Hn, #fbits ; Half-precision to 32-bit
FCVTZS Xd, Hn, #fbits ; Half-precision to 64-bit
FCVTZS Wd, Sn, #fbits ; Single-precision to 32-bit
FCVTZS Xd, Sn, #fbits ; Single-precision to 64-bit
FCVTZS Wd, Dn, #fbits ; Double-precision to 32-bit
FCVTZS Xd, Dn, #fbits ; Double-precision to 64-bit
```

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*fbits*

Depends on the instruction variant:

**32-bit**

Is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32.

**64-bit**

Is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = signed\_convertToIntegerExactTowardZero(*Vn*\*(2<sup>*fbits*</sup>)), where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.20 FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar).

### Syntax

```
FCVTZS Wd, Hn ; Half-precision to 32-bit
FCVTZS Xd, Hn ; Half-precision to 64-bit
FCVTZS Wd, Sn ; Single-precision to 32-bit
FCVTZS Xd, Sn ; Single-precision to 64-bit
FCVTZS Wd, Dn ; Double-precision to 32-bit
FCVTZS Xd, Dn ; Double-precision to 64-bit
```

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = signed\_convertToIntegerExactTowardZero(*Vn*), where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.21 FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).

### Syntax

```
FCVTZU Wd, Hn, #fbits ; Half-precision to 32-bit
FCVTZU Xd, Hn, #fbits ; Half-precision to 64-bit
FCVTZU Wd, Sn, #fbits ; Single-precision to 32-bit
FCVTZU Xd, Sn, #fbits ; Single-precision to 64-bit
FCVTZU Wd, Dn, #fbits ; Double-precision to 32-bit
FCVTZU Xd, Dn, #fbits ; Double-precision to 64-bit
```

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*fbits*

Depends on the instruction variant:

**32-bit**

Is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32.

**64-bit**

Is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `unsigned_convertToIntegerExactTowardZero(Vn*(2^fbits))`, where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.22 FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).

### Syntax

```
FCVTZU Wd, Hn ; Half-precision to 32-bit  
FCVTZU Xd, Hn ; Half-precision to 64-bit  
FCVTZU Wd, Sn ; Single-precision to 32-bit  
FCVTZU Xd, Sn ; Single-precision to 64-bit  
FCVTZU Wd, Dn ; Double-precision to 32-bit  
FCVTZU Xd, Dn ; Double-precision to 64-bit
```

Where:

- Wd* Is the 32-bit name of the general-purpose destination register.
- Hn* Is the 16-bit name of the SIMD and FP source register.
- Xd* Is the 64-bit name of the general-purpose destination register.
- Sn* Is the 32-bit name of the SIMD and FP source register.
- Dn* Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Rd* = `unsigned_convertToIntegerExactTowardZero(Vn)`, where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.23 FDIV (scalar)

Floating-point Divide (scalar).

### Syntax

FDIV *Hd*, *Hn*, *Hm* ; Half-precision

FDIV *Sd*, *Sn*, *Sm* ; Single-precision

FDIV *Dd*, *Dn*, *Dm* ; Double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD and FP register by the floating-point value of the second source SIMD and FP register, and writes the result to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

$$V_d = V_n / V_m.$$

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.24 FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.

### Syntax

`FJCVTZS Wd, Dn`

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Architectures supported

Supported in Armv8.3-A architecture and later.

### Usage

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD and FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and write the result to the general-purpose destination register. If the result is too large to be held as a 32-bit signed integer, then the result is the integer modulo  $2^{32}$ , as held in a 32-bit signed integer.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.25 FMADD

Floating-point fused Multiply-Add (scalar).

### Syntax

```
FMADD Hd, Hn, Hm, Ha ; Half-precision  
FMADD Sd, Sn, Sm, Sa ; Single-precision  
FMADD Dd, Dn, Dm, Da ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register holding the multiplicand.

*Hm*

Is the 16-bit name of the second SIMD and FP source register holding the multiplier.

*Ha*

Is the 16-bit name of the third SIMD and FP source register holding the addend.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register holding the multiplicand.

*Sm*

Is the 32-bit name of the second SIMD and FP source register holding the multiplier.

*Sa*

Is the 32-bit name of the third SIMD and FP source register holding the addend.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register holding the multiplicand.

*Dm*

Is the 64-bit name of the second SIMD and FP source register holding the multiplier.

*Da*

Is the 64-bit name of the third SIMD and FP source register holding the addend.

### Operation

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD and FP source registers, adds the product to the value of the third SIMD and FP source register, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

$$Vd = Va + Vn \cdot Vm$$

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.26 FMAX (scalar)

Floating-point Maximum (scalar).

### Syntax

FMAX *Hd*, *Hn*, *Hm* ; Half-precision

FMAX *Sd*, *Sn*, *Sm* ; Single-precision

FMAX *Dd*, *Dn*, *Dm* ; Double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Maximum (scalar). This instruction compares the two source SIMD and FP registers, and writes the larger of the two floating-point values to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = max(*Vn*, *Vm*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.27 FMAXNM (scalar)

Floating-point Maximum Number (scalar).

### Syntax

```
FMAXNM Hd, Hn, Hm ; Half-precision  
FMAXNM Sd, Sn, Sm ; Single-precision  
FMAXNM Dd, Dn, Dm ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD and FP register values, and writes the larger of the two floating-point values to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

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.

*Vd* = maxNum(*Vn*, *Vm*).

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.28 FMIN (scalar)

Floating-point Minimum (scalar).

### Syntax

FMIN *Hd*, *Hn*, *Hm* ; Half-precision

FMIN *Sd*, *Sn*, *Sm* ; Single-precision

FMIN *Dd*, *Dn*, *Dm* ; Double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD and FP register values, and writes the smaller of the two floating-point values to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = min(*Vn*, *Vm*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.29 FMINNM (scalar)

Floating-point Minimum Number (scalar).

### Syntax

```
FMINNM Hd, Hn, Hm ; Half-precision  
FMINNM Sd, Sn, Sm ; Single-precision  
FMINNM Dd, Dn, Dm ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD and FP register values, and writes the smaller of the two floating-point values to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = minNum(*Vn*, *Vm*).

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.30 FMOV (register)

Floating-point Move register without conversion.

### Syntax

```
FMOV Hd, Hn ; Half-precision  
FMOV Sd, Sn ; Single-precision  
FMOV Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD and FP source register to the SIMD and 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.

$Vd = Vn$ .

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

## D4.31 FMOV (general)

Floating-point Move to or from general-purpose register without conversion.

### Syntax

```
FMOV Wd, Hn ; Half-precision to 32-bit  
FMOV Xd, Hn ; Half-precision to 64-bit  
FMOV Hd, Wn ; 32-bit to half-precision  
FMOV Sd, Wn ; 32-bit to single-precision  
FMOV Wd, Sn ; Single-precision to 32-bit  
FMOV Hd, Xn ; 64-bit to half-precision  
FMOV Dd, Xn ; 64-bit to double-precision  
FMOV Vd.D[1], Xn ; 64-bit to top half of 128-bit  
FMOV Xd, Dn ; Double-precision to 64-bit  
FMOV Xd, Vn.D[1] ; Top half of 128-bit to 64-bit
```

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Xn*

Is the 64-bit name of the general-purpose source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Vd*

Is the name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD and FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD and 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.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

## D4.32 FMOV (scalar, immediate)

Floating-point move immediate (scalar).

### Syntax

```
FMOV Hd, #imm ; Half-precision  
FMOV Sd, #imm ; Single-precision  
FMOV Dd, #imm ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*imm*

Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision. For details of the range of constants available and the encoding of *imm*, see *Modified immediate constants in A64 floating-point instructions* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

### Operation

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD and 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.

*Vd*=#*imm*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.33 FMSUB

Floating-point Fused Multiply-Subtract (scalar).

### Syntax

```
FMSUB Hd, Hn, Hm, Ha ; Half-precision
FMSUB Sd, Sn, Sm, Sa ; Single-precision
FMSUB Dd, Dn, Dm, Da ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register holding the multiplicand.

*Hm*

Is the 16-bit name of the second SIMD and FP source register holding the multiplier.

*Ha*

Is the 16-bit name of the third SIMD and FP source register holding the minuend.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register holding the multiplicand.

*Sm*

Is the 32-bit name of the second SIMD and FP source register holding the multiplier.

*Sa*

Is the 32-bit name of the third SIMD and FP source register holding the minuend.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register holding the multiplicand.

*Dm*

Is the 64-bit name of the second SIMD and FP source register holding the multiplier.

*Da*

Is the 64-bit name of the third SIMD and FP source register holding the minuend.

### Operation

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD and FP source registers, negates the product, adds that to the value of the third SIMD and FP source register, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

$$Vd = Va + (-Vn) * Vm.$$

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.34 FMUL (scalar)

Floating-point Multiply (scalar).

### Syntax

```
FMUL Hd, Hn, Hm ; Half-precision  
FMUL Sd, Sn, Sm ; Single-precision  
FMUL Dd, Dn, Dm ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD and FP registers, and writes the result to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

$V_d = V_n * V_m$ .

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.35 FNEG (scalar)

Floating-point Negate (scalar).

### Syntax

FNEG *Hd*, *Hn* ; Half-precision  
FNEG *Sd*, *Sn* ; Single-precision  
FNEG *Dd*, *Dn* ; Double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Negate (scalar). This instruction negates the value in the SIMD and FP source register and writes the result to the SIMD and 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.

$V_d = -V_n$ .

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

## D4.36 FNMADD

Floating-point Negated fused Multiply-Add (scalar).

### Syntax

```
FNMADD Hd, Hn, Hm, Ha ; Half-precision
FNMADD Sd, Sn, Sm, Sa ; Single-precision
FNMADD Dd, Dn, Dm, Da ; Double-precision
```

Where:

<b>Hd</b>	Is the 16-bit name of the SIMD and FP destination register.
<b>Hn</b>	Is the 16-bit name of the first SIMD and FP source register holding the multiplicand.
<b>Hm</b>	Is the 16-bit name of the second SIMD and FP source register holding the multiplier.
<b>Ha</b>	Is the 16-bit name of the third SIMD and FP source register holding the addend.
<b>Sd</b>	Is the 32-bit name of the SIMD and FP destination register.
<b>Sn</b>	Is the 32-bit name of the first SIMD and FP source register holding the multiplicand.
<b>Sm</b>	Is the 32-bit name of the second SIMD and FP source register holding the multiplier.
<b>Sa</b>	Is the 32-bit name of the third SIMD and FP source register holding the addend.
<b>Dd</b>	Is the 64-bit name of the SIMD and FP destination register.
<b>Dn</b>	Is the 64-bit name of the first SIMD and FP source register holding the multiplicand.
<b>Dm</b>	Is the 64-bit name of the second SIMD and FP source register holding the multiplier.
<b>Da</b>	Is the 64-bit name of the third SIMD and FP source register holding the addend.

### Operation

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD and FP source registers, negates the product, subtracts the value of the third SIMD and FP source register, and writes the result to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

$$Vd = (-Vn) + (-Vm) * Vm.$$

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.37 FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar).

### Syntax

```
FNMSUB Hd, Hn, Hm, Ha ; Half-precision
FNMSUB Sd, Sn, Sm, Sa ; Single-precision
FNMSUB Dd, Dn, Dm, Da ; Double-precision
```

Where:

**Hd**

Is the 16-bit name of the SIMD and FP destination register.

**Hn**

Is the 16-bit name of the first SIMD and FP source register holding the multiplicand.

**Hm**

Is the 16-bit name of the second SIMD and FP source register holding the multiplier.

**Ha**

Is the 16-bit name of the third SIMD and FP source register holding the minuend.

**Sd**

Is the 32-bit name of the SIMD and FP destination register.

**Sn**

Is the 32-bit name of the first SIMD and FP source register holding the multiplicand.

**Sm**

Is the 32-bit name of the second SIMD and FP source register holding the multiplier.

**Sa**

Is the 32-bit name of the third SIMD and FP source register holding the minuend.

**Dd**

Is the 64-bit name of the SIMD and FP destination register.

**Dn**

Is the 64-bit name of the first SIMD and FP source register holding the multiplicand.

**Dm**

Is the 64-bit name of the second SIMD and FP source register holding the multiplier.

**Da**

Is the 64-bit name of the third SIMD and FP source register holding the minuend.

### Operation

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD and FP source registers, subtracts the value of the third SIMD and FP source register, and writes the result to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

$$Vd = (-Va) + Vn \cdot Vm$$

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.38 FNMUL (scalar)

Floating-point Multiply-Negate (scalar).

### Syntax

FNMUL *Hd*, *Hn*, *Hm* ; Half-precision

FNMUL *Sd*, *Sn*, *Sm* ; Single-precision

FNMUL *Dd*, *Dn*, *Dm* ; Double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD and FP registers, and writes the negation of the result to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

$$V_d = -(V_n * V_m).$$

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.39 FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar).

### Syntax

```
FRINTA Hd, Hn ; Half-precision  
FRINTA Sd, Sn ; Single-precision  
FRINTA Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD and 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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = roundToIntegralTiesToAway(*Vn*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.40 FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar).

### Syntax

```
FRINTI Hd, Hn ; Half-precision  
FRINTI Sd, Sn ; Single-precision  
FRINTI Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD and 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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = roundToIntegral(*Vn*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.41 FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar).

### Syntax

```
FRINTM Hd, Hn ; Half-precision  
FRINTM Sd, Sn ; Single-precision  
FRINTM Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD and 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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = roundToIntegralTowardNegative(*Vn*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.42 FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar).

### Syntax

```
FRINTN Hd, Hn ; Half-precision  
FRINTN Sd, Sn ; Single-precision  
FRINTN Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD and 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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = roundToIntegralTiesToEven(*Vn*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.43 FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar).

### Syntax

```
FRINTP Hd, Hn ; Half-precision  
FRINTP Sd, Sn ; Single-precision  
FRINTP Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD and 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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = roundToIntegralTowardPositive(*Vn*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.44 FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar).

### Syntax

```
FRINTX Hd, Hn ; Half-precision  
FRINTX Sd, Sn ; Single-precision  
FRINTX Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD and 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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = roundToIntegralExact(*Vn*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.45 FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar).

### Syntax

```
FRINTZ Hd, Hn ; Half-precision  
FRINTZ Sd, Sn ; Single-precision  
FRINTZ Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD and 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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = roundToIntegralTowardZero(*Vn*).

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.46 FSQRT (scalar)

Floating-point Square Root (scalar).

### Syntax

```
FSQRT Hd, Hn ; Half-precision  
FSQRT Sd, Sn ; Single-precision  
FSQRT Dd, Dn ; Double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the SIMD and FP source register.

### Operation

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD and FP source register and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = *sqrt(Vn)*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.47 FSUB (scalar)

Floating-point Subtract (scalar).

### Syntax

FSUB *Hd*, *Hn*, *Hm* ; Half-precision

FSUB *Sd*, *Sn*, *Sm* ; Single-precision

FSUB *Dd*, *Dn*, *Dm* ; Double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Sn*

Is the 32-bit name of the first SIMD and FP source register.

*Sm*

Is the 32-bit name of the second SIMD and FP source register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Dn*

Is the 64-bit name of the first SIMD and FP source register.

*Dm*

Is the 64-bit name of the second SIMD and FP source register.

### Operation

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD and FP register from the floating-point value of the first source SIMD and FP register, and writes the result to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

$$V_d = V_n - V_m.$$

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.48 LDNP (SIMD and FP)

Load Pair of SIMD and FP registers, with Non-temporal hint.

### Syntax

```
LDNP St1, St2, [Xn/SP{, #imm}] ; 32-bit  
LDNP Dt1, Dt2, [Xn/SP{, #imm}] ; 64-bit  
LDNP Qt1, Qt2, [Xn/SP{, #imm}] ; 128-bit
```

Where:

#### St1

Is the 32-bit name of the first SIMD and FP register to be transferred.

#### St2

Is the 32-bit name of the second SIMD and FP register to be transferred.

#### imm

Depends on the instruction variant:

#### 32-bit

Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

#### 64-bit

Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0.

#### 128-bit

Is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0.

#### Dt1

Is the 64-bit name of the first SIMD and FP register to be transferred.

#### Dt2

Is the 64-bit name of the second SIMD and FP register to be transferred.

#### Qt1

Is the 128-bit name of the first SIMD and FP register to be transferred.

#### Qt2

Is the 128-bit name of the second SIMD and FP register to be transferred.

#### Xn/SP

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

Load Pair of SIMD and FP registers, with Non-temporal hint. This instruction loads a pair of SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

#### Note

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDNP (SIMD and FP).

**Related reference**

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.49 LDP (SIMD and FP)

Load Pair of SIMD and FP registers.

### Syntax

```
LDP St1, St2, [Xn/SP], #imm ; 32-bit, Post-index  
LDP Dt1, Dt2, [Xn/SP], #imm ; 64-bit, Post-index  
LDP Qt1, Qt2, [Xn/SP], #imm ; 128-bit, Post-index  
LDP St1, St2, [Xn/SP, #imm]! ; 32-bit, Pre-index  
LDP Dt1, Dt2, [Xn/SP, #imm]! ; 64-bit, Pre-index  
LDP Qt1, Qt2, [Xn/SP, #imm]! ; 128-bit, Pre-index  
LDP St1, St2, [Xn/SP{, #imm}] ; 32-bit, Signed offset  
LDP Dt1, Dt2, [Xn/SP{, #imm}] ; 64-bit, Signed offset  
LDP Qt1, Qt2, [Xn/SP{, #imm}] ; 128-bit, Signed offset
```

Where:

**St1**

Is the 32-bit name of the first SIMD and FP register to be transferred.

**St2**

Is the 32-bit name of the second SIMD and FP register to be transferred.

**imm**

Depends on the instruction variant:

**32-bit**

Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

**64-bit**

Is the signed immediate byte offset, a multiple of 8 in the range -512 to 504.

**128-bit**

Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008.

**Dt1**

Is the 64-bit name of the first SIMD and FP register to be transferred.

**Dt2**

Is the 64-bit name of the second SIMD and FP register to be transferred.

**Qt1**

Is the 128-bit name of the first SIMD and FP register to be transferred.

**Qt2**

Is the 128-bit name of the second SIMD and FP register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

Load Pair of SIMD and FP registers. This instruction loads a pair of SIMD and 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.

———— **Note** ————

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#), and particularly LDP (SIMD and FP).

**Related reference**

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

**Related information**

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.50 LDR (immediate, SIMD and FP)

Load SIMD and FP Register (immediate offset).

### Syntax

```
LDR <Bt>, [Xn/SP], #simm ; 8-bit, Post-index
LDR Ht, [Xn/SP], #simm ; 16-bit, Post-index
LDR St, [Xn/SP], #simm ; 32-bit, Post-index
LDR Dt, [Xn/SP], #simm ; 64-bit, Post-index
LDR Qt, [Xn/SP], #simm ; 128-bit, Post-index
LDR <Bt>, [Xn/SP, #simm]! ; 8-bit, Pre-index
LDR Ht, [Xn/SP, #simm]! ; 16-bit, Pre-index
LDR St, [Xn/SP, #simm]! ; 32-bit, Pre-index
LDR Dt, [Xn/SP, #simm]! ; 64-bit, Pre-index
LDR Qt, [Xn/SP, #simm]! ; 128-bit, Pre-index
LDR <Bt>, [Xn/SP{, #pimm}] ; 8-bit, Unsigned offset
LDR Ht, [Xn/SP{, #pimm}] ; 16-bit, Unsigned offset
LDR St, [Xn/SP{, #pimm}] ; 32-bit, Unsigned offset
LDR Dt, [Xn/SP{, #pimm}] ; 64-bit, Unsigned offset
LDR Qt, [Xn/SP{, #pimm}] ; 128-bit, Unsigned offset
```

Where:

**<Bt>**

Is the 8-bit name of the SIMD and FP register to be transferred.

**simm**

Is the signed immediate byte offset, in the range -256 to 255.

**Ht**

Is the 16-bit name of the SIMD and FP register to be transferred.

**St**

Is the 32-bit name of the SIMD and FP register to be transferred.

**Dt**

Is the 64-bit name of the SIMD and FP register to be transferred.

**Qt**

Is the 128-bit name of the SIMD and FP register to be transferred.

**pimm**

Depends on the instruction variant:

**8-bit**

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

**16-bit**

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

**32-bit**

Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

**64-bit**

Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

**128-bit**

Is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0.

***Xn/SP***

Is the 64-bit name of the general-purpose base register or stack pointer.

**Usage**

Load SIMD and FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD and 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.

**Related reference**

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

## D4.51 LDR (literal, SIMD and FP)

Load SIMD and FP Register (PC-relative literal).

### Syntax

LDR *St*, *Label* ; 32-bit

LDR *Dt*, *Label* ; 64-bit

LDR *Qt*, *Label* ; 128-bit

Where:

***St***

Is the 32-bit name of the SIMD and FP register to be loaded.

***Dt***

Is the 64-bit name of the SIMD and FP register to be loaded.

***Qt***

Is the 128-bit name of the SIMD and FP register to be loaded.

***Label***

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range  $\pm 1\text{MB}$ .

### Usage

Load SIMD and FP Register (PC-relative literal). This instruction loads a SIMD and FP register from memory. The address that is used for the load is calculated from the PC 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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

## D4.52 LDR (register, SIMD and FP)

Load SIMD and FP Register (register offset).

### Syntax

```
LDR <Bt>, [Xn/SP, (Wm|Xm), extend {amount}] ; 8-bit
LDR <Bt>, [Xn/SP, Xm{, LSL amount}] ; 8-bit
LDR Ht, [Xn/SP, (Wm|Xm){, extend {amount}}] ; 16-bit
LDR St, [Xn/SP, (Wm|Xm){, extend {amount}}] ; 32-bit
LDR Dt, [Xn/SP, (Wm|Xm){, extend {amount}}] ; 64-bit
LDR Qt, [Xn/SP, (Wm|Xm){, extend {amount}}] ; 128-bit
```

Where:

**<Bt>**

Is the 8-bit name of the SIMD and FP register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Wm**

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

**Xm**

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

**extend**

Is the index extend specifier:

**8-bit**

Can be one of UXTW, SXTW or SXTX.

**16-bit, 32-bit, 64-bit, and 128-bit**

Can be one of UXTW, LSL, SXTW or SXTX. LSL is the default, and must be omitted for the LSL option when *amount* is omitted.

**amount**

Is the index shift amount:

**8-bit**

Must be #0.

**16-bit**

Can be #0 or #1. Optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0.

**32-bit**

Can be #0 or #2. Optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0.

**64-bit**

Can be #0 or #3. Optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0.

**128-bit**

Can be #0 or #4. Optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0.

**Ht**

Is the 16-bit name of the SIMD and FP register to be transferred.

**St**

Is the 32-bit name of the SIMD and FP register to be transferred.

***Dt***

Is the 64-bit name of the SIMD and FP register to be transferred.

***Qt***

Is the 128-bit name of the SIMD and FP register to be transferred.

### Usage

Load SIMD and FP Register (register offset). This instruction loads a SIMD and 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, CPTTR\_EL2, and CPTTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### *Related reference*

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

## D4.53 LDUR (SIMD and FP)

Load SIMD and FP Register (unscaled offset).

### Syntax

```
LDUR <Bt>, [Xn/SP{, #simm}] ; 8-bit  
LDUR Ht, [Xn/SP{, #simm}] ; 16-bit  
LDUR St, [Xn/SP{, #simm}] ; 32-bit  
LDUR Dt, [Xn/SP{, #simm}] ; 64-bit  
LDUR Qt, [Xn/SP{, #simm}] ; 128-bit
```

Where:

**<Bt>**

Is the 8-bit name of the SIMD and FP register to be transferred.

**Ht**

Is the 16-bit name of the SIMD and FP register to be transferred.

**St**

Is the 32-bit name of the SIMD and FP register to be transferred.

**Dt**

Is the 64-bit name of the SIMD and FP register to be transferred.

**Qt**

Is the 128-bit name of the SIMD and FP register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**simm**

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

Load SIMD and FP Register (unscaled offset). This instruction loads a SIMD and 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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

## D4.54 SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar).

### Syntax

```
SCVTF Hd, Wn, #fbits ; 32-bit to half-precision
SCVTF Sd, Wn, #fbits ; 32-bit to single-precision
SCVTF Dd, Wn, #fbits ; 32-bit to double-precision
SCVTF Hd, Xn, #fbits ; 64-bit to half-precision
SCVTF Sd, Xn, #fbits ; 64-bit to single-precision
SCVTF Dd, Xn, #fbits ; 64-bit to double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*fbits*

Depends on the instruction variant:

#### 32-bit

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.

#### 64-bit

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.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = `signed_convertFromInt(Rn/(2fbits))`, where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.55 SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar).

### Syntax

```
SCVTF Hd, Wn ; 32-bit to half-precision  
SCVTF Sd, Wn ; 32-bit to single-precision  
SCVTF Dd, Wn ; 32-bit to double-precision  
SCVTF Hd, Xn ; 64-bit to half-precision  
SCVTF Sd, Xn ; 64-bit to single-precision  
SCVTF Dd, Xn ; 64-bit to double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = signed\_convertFromInt(*Rn*), where *R* is either *W* or *X*.

### Related reference

[D4.1 A64 floating-point instructions in alphabetical order](#) on page D4-1029

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.56 STNP (SIMD and FP)

Store Pair of SIMD and FP registers, with Non-temporal hint.

### Syntax

```
STNP St1, St2, [Xn/SP{, #imm}] ; 32-bit  
STNP Dt1, Dt2, [Xn/SP{, #imm}] ; 64-bit  
STNP Qt1, Qt2, [Xn/SP{, #imm}] ; 128-bit
```

Where:

#### *St1*

Is the 32-bit name of the first SIMD and FP register to be transferred.

#### *St2*

Is the 32-bit name of the second SIMD and FP register to be transferred.

#### *imm*

Depends on the instruction variant:

#### 32-bit

Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

#### 64-bit

Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0.

#### 128-bit

Is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0.

#### *Dt1*

Is the 64-bit name of the first SIMD and FP register to be transferred.

#### *Dt2*

Is the 64-bit name of the second SIMD and FP register to be transferred.

#### *Qt1*

Is the 128-bit name of the first SIMD and FP register to be transferred.

#### *Qt2*

Is the 128-bit name of the second SIMD and FP register to be transferred.

#### *Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

Store Pair of SIMD and FP registers, with Non-temporal hint. This instruction stores a pair of SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D4.57 STP (SIMD and FP)

Store Pair of SIMD and FP registers.

### Syntax

```
STP St1, St2, [Xn/SP], #imm ; 32-bit, Post-index  
STP Dt1, Dt2, [Xn/SP], #imm ; 64-bit, Post-index  
STP Qt1, Qt2, [Xn/SP], #imm ; 128-bit, Post-index  
STP St1, St2, [Xn/SP, #imm]! ; 32-bit, Pre-index  
STP Dt1, Dt2, [Xn/SP, #imm]! ; 64-bit, Pre-index  
STP Qt1, Qt2, [Xn/SP, #imm]! ; 128-bit, Pre-index  
STP St1, St2, [Xn/SP{, #imm}] ; 32-bit, Signed offset  
STP Dt1, Dt2, [Xn/SP{, #imm}] ; 64-bit, Signed offset  
STP Qt1, Qt2, [Xn/SP{, #imm}] ; 128-bit, Signed offset
```

Where:

**St1**

Is the 32-bit name of the first SIMD and FP register to be transferred.

**St2**

Is the 32-bit name of the second SIMD and FP register to be transferred.

**imm**

Depends on the instruction variant:

**32-bit**

Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

**64-bit**

Is the signed immediate byte offset, a multiple of 8 in the range -512 to 504.

**128-bit**

Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008.

**Dt1**

Is the 64-bit name of the first SIMD and FP register to be transferred.

**Dt2**

Is the 64-bit name of the second SIMD and FP register to be transferred.

**Qt1**

Is the 128-bit name of the first SIMD and FP register to be transferred.

**Qt2**

Is the 128-bit name of the second SIMD and FP register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

Store Pair of SIMD and FP registers. This instruction stores a pair of SIMD and 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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

## D4.58 STR (immediate, SIMD and FP)

Store SIMD and FP register (immediate offset).

### Syntax

```

STR <Bt>, [Xn/SP], #simm ; 8-bit, Post-index
STR Ht, [Xn/SP], #simm ; 16-bit, Post-index
STR St, [Xn/SP], #simm ; 32-bit, Post-index
STR Dt, [Xn/SP], #simm ; 64-bit, Post-index
STR Qt, [Xn/SP], #simm ; 128-bit, Post-index
STR <Bt>, [Xn/SP, #simm]! ; 8-bit, Pre-index
STR Ht, [Xn/SP, #simm]! ; 16-bit, Pre-index
STR St, [Xn/SP, #simm]! ; 32-bit, Pre-index
STR Dt, [Xn/SP, #simm]! ; 64-bit, Pre-index
STR Qt, [Xn/SP, #simm]! ; 128-bit, Pre-index
STR <Bt>, [Xn/SP{, #pimm}] ; 8-bit, Unsigned offset
STR Ht, [Xn/SP{, #pimm}] ; 16-bit, Unsigned offset
STR St, [Xn/SP{, #pimm}] ; 32-bit, Unsigned offset
STR Dt, [Xn/SP{, #pimm}] ; 64-bit, Unsigned offset
STR Qt, [Xn/SP{, #pimm}] ; 128-bit, Unsigned offset

```

Where:

**<Bt>**

Is the 8-bit name of the SIMD and FP register to be transferred.

**simm**

Is the signed immediate byte offset, in the range -256 to 255.

**Ht**

Is the 16-bit name of the SIMD and FP register to be transferred.

**St**

Is the 32-bit name of the SIMD and FP register to be transferred.

**Dt**

Is the 64-bit name of the SIMD and FP register to be transferred.

**Qt**

Is the 128-bit name of the SIMD and FP register to be transferred.

**pimm**

Depends on the instruction variant:

**8-bit**

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

**16-bit**

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

**32-bit**

Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

**64-bit**

Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

**128-bit**

Is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Usage**

Store SIMD and FP register (immediate offset). This instruction stores a single SIMD and 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.

**Related reference**

*D3.1 A64 data transfer instructions in alphabetical order* on page D3-877

## D4.59 STR (register, SIMD and FP)

Store SIMD and FP register (register offset).

### Syntax

```
STR <Bt>, [Xn/SP, (Wm|Xm), extend {amount}] ; 8-bit
STR <Bt>, [Xn/SP, Xm{, LSL amount}] ; 8-bit
STR Ht, [Xn/SP, (Wm|Xm){, extend {amount}}] ; 16-bit
STR St, [Xn/SP, (Wm|Xm){, extend {amount}}] ; 32-bit
STR Dt, [Xn/SP, (Wm|Xm){, extend {amount}}] ; 64-bit
STR Qt, [Xn/SP, (Wm|Xm){, extend {amount}}] ; 128-bit
```

Where:

**<Bt>**

Is the 8-bit name of the SIMD and FP register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Wm**

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

**Xm**

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

**extend**

Is the index extend specifier:

**8-bit**

Can be one of UXTW, SXTW or SXTX.

**16-bit, 32-bit, 64-bit, and 128-bit**

Can be one of UXTW, LSL, SXTW or SXTX. LSL is the default, and must be omitted for the LSL option when *amount* is omitted.

**amount**

Is the index shift amount:

**8-bit**

Must be #0.

**16-bit**

Can be #0 or #1. Optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0.

**32-bit**

Can be #0 or #2. Optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0.

**64-bit**

Can be #0 or #3. Optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0.

**128-bit**

Can be #0 or #4. Optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0.

**Ht**

Is the 16-bit name of the SIMD and FP register to be transferred.

**St**

Is the 32-bit name of the SIMD and FP register to be transferred.

***Dt***

Is the 64-bit name of the SIMD and FP register to be transferred.

***Qt***

Is the 128-bit name of the SIMD and FP register to be transferred.

### Usage

Store SIMD and FP register (register offset). This instruction stores a single SIMD and 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, CPTTR\_EL2, and CPTTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### *Related reference*

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

## D4.60 STUR (SIMD and FP)

Store SIMD and FP register (unscaled offset).

### Syntax

```
STUR <Bt>, [Xn/SP{, #simm}] ; 8-bit  
STUR Ht, [Xn/SP{, #simm}] ; 16-bit  
STUR St, [Xn/SP{, #simm}] ; 32-bit  
STUR Dt, [Xn/SP{, #simm}] ; 64-bit  
STUR Qt, [Xn/SP{, #simm}] ; 128-bit
```

Where:

**<Bt>**

Is the 8-bit name of the SIMD and FP register to be transferred.

**Ht**

Is the 16-bit name of the SIMD and FP register to be transferred.

**St**

Is the 32-bit name of the SIMD and FP register to be transferred.

**Dt**

Is the 64-bit name of the SIMD and FP register to be transferred.

**Qt**

Is the 128-bit name of the SIMD and FP register to be transferred.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**simm**

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

### Usage

Store SIMD and FP register (unscaled offset). This instruction stores a single SIMD and 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.

### Related reference

[D3.1 A64 data transfer instructions in alphabetical order](#) on page D3-877

## D4.61 UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar).

### Syntax

```
UCVTF Hd, Wn, #fbits ; 32-bit to half-precision  
UCVTF Sd, Wn, #fbits ; 32-bit to single-precision  
UCVTF Dd, Wn, #fbits ; 32-bit to double-precision  
UCVTF Hd, Xn, #fbits ; 64-bit to half-precision  
UCVTF Sd, Xn, #fbits ; 64-bit to single-precision  
UCVTF Dd, Xn, #fbits ; 64-bit to double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Wn*

Is the 32-bit name of the general-purpose source register.

*fbits*

Depends on the instruction variant:

#### 32-bit

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.

#### 64-bit

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.

*Sd*

Is the 32-bit name of the SIMD and FP destination register.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*Xn*

Is the 64-bit name of the general-purpose source register.

### Operation

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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd* = `unsigned_convertFromInt(Rn/(2^fbits))`, where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D4.62 UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar).

### Syntax

```
UCVTF Hd, Wn ; 32-bit to half-precision  
UCVTF Sd, Wn ; 32-bit to single-precision  
UCVTF Dd, Wn ; 32-bit to double-precision  
UCVTF Hd, Xn ; 64-bit to half-precision  
UCVTF Sd, Xn ; 64-bit to single-precision  
UCVTF Dd, Xn ; 64-bit to double-precision
```

Where:

**Hd** Is the 16-bit name of the SIMD and FP destination register.  
**Wn** Is the 32-bit name of the general-purpose source register.  
**Sd** Is the 32-bit name of the SIMD and FP destination register.  
**Dd** Is the 64-bit name of the SIMD and FP destination register.  
**Xn** Is the 64-bit name of the general-purpose source register.

### Operation

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 and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

*Vd = unsigned\_convertFromInt(Rn)*, where *R* is either *W* or *X*.

### Related reference

*D4.1 A64 floating-point instructions in alphabetical order* on page D4-1029

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*



# Chapter D5

## A64 SIMD Scalar Instructions

Describes the A64 SIMD scalar instructions.

It contains the following sections:

- [\*D5.1 A64 SIMD scalar instructions in alphabetical order\*](#) on page D5-1110.
- [\*D5.2 ABS \(scalar\)\*](#) on page D5-1115.
- [\*D5.3 ADD \(scalar\)\*](#) on page D5-1116.
- [\*D5.4 ADDP \(scalar\)\*](#) on page D5-1117.
- [\*D5.5 CMEQ \(scalar, register\)\*](#) on page D5-1118.
- [\*D5.6 CMEQ \(scalar, zero\)\*](#) on page D5-1119.
- [\*D5.7 CMGE \(scalar, register\)\*](#) on page D5-1120.
- [\*D5.8 CMGE \(scalar, zero\)\*](#) on page D5-1121.
- [\*D5.9 CMGT \(scalar, register\)\*](#) on page D5-1122.
- [\*D5.10 CMGT \(scalar, zero\)\*](#) on page D5-1123.
- [\*D5.11 CMHI \(scalar, register\)\*](#) on page D5-1124.
- [\*D5.12 CMHS \(scalar, register\)\*](#) on page D5-1125.
- [\*D5.13 CMLE \(scalar, zero\)\*](#) on page D5-1126.
- [\*D5.14 CMLT \(scalar, zero\)\*](#) on page D5-1127.
- [\*D5.15 CMTST \(scalar\)\*](#) on page D5-1128.
- [\*D5.16 DUP \(scalar, element\)\*](#) on page D5-1129.
- [\*D5.17 FABD \(scalar\)\*](#) on page D5-1130.
- [\*D5.18 FACGE \(scalar\)\*](#) on page D5-1131.
- [\*D5.19 FACGT \(scalar\)\*](#) on page D5-1132.
- [\*D5.20 FADDP \(scalar\)\*](#) on page D5-1133.
- [\*D5.21 FCMEQ \(scalar, register\)\*](#) on page D5-1134.
- [\*D5.22 FCMEQ \(scalar, zero\)\*](#) on page D5-1135.
- [\*D5.23 FCMGE \(scalar, register\)\*](#) on page D5-1136.

- [D5.24 FCMGE \(scalar, zero\)](#) on page D5-1137.
- [D5.25 FCMGT \(scalar, register\)](#) on page D5-1138.
- [D5.26 FCMGT \(scalar, zero\)](#) on page D5-1139.
- [D5.27 FCMLA \(scalar, by element\)](#) on page D5-1140.
- [D5.28 FCMLE \(scalar, zero\)](#) on page D5-1142.
- [D5.29 FCMLT \(scalar, zero\)](#) on page D5-1143.
- [D5.30 FCVTAS \(scalar\)](#) on page D5-1144.
- [D5.31 FCVTAU \(scalar\)](#) on page D5-1145.
- [D5.32 FCVTMS \(scalar\)](#) on page D5-1146.
- [D5.33 FCVTMU \(scalar\)](#) on page D5-1147.
- [D5.34 FCVTNS \(scalar\)](#) on page D5-1148.
- [D5.35 FCVTNU \(scalar\)](#) on page D5-1149.
- [D5.36 FCVTPS \(scalar\)](#) on page D5-1150.
- [D5.37 FCVTPU \(scalar\)](#) on page D5-1151.
- [D5.38 FCVTXN \(scalar\)](#) on page D5-1152.
- [D5.39 FCVTZS \(scalar, fixed-point\)](#) on page D5-1153.
- [D5.40 FCVTZS \(scalar, integer\)](#) on page D5-1154.
- [D5.41 FCVTZU \(scalar, fixed-point\)](#) on page D5-1155.
- [D5.42 FCVTZU \(scalar, integer\)](#) on page D5-1156.
- [D5.43 FMAXNMP \(scalar\)](#) on page D5-1157.
- [D5.44 FMAXP \(scalar\)](#) on page D5-1158.
- [D5.45 FMINNMP \(scalar\)](#) on page D5-1159.
- [D5.46 FMINP \(scalar\)](#) on page D5-1160.
- [D5.47 FMLA \(scalar, by element\)](#) on page D5-1161.
- [D5.48 FMLAL, \(scalar, by element\)](#) on page D5-1163.
- [D5.49 FMLS \(scalar, by element\)](#) on page D5-1164.
- [D5.50 FMLSL, \(scalar, by element\)](#) on page D5-1166.
- [D5.51 FMUL \(scalar, by element\)](#) on page D5-1167.
- [D5.52 FMULX \(scalar, by element\)](#) on page D5-1169.
- [D5.53 FMULX \(scalar\)](#) on page D5-1171.
- [D5.54 FRECPE \(scalar\)](#) on page D5-1172.
- [D5.55 FRECPSS \(scalar\)](#) on page D5-1173.
- [D5.56 FRSQRTE \(scalar\)](#) on page D5-1174.
- [D5.57 FRSQRTS \(scalar\)](#) on page D5-1175.
- [D5.58 MOV \(scalar\)](#) on page D5-1176.
- [D5.59 NEG \(scalar\)](#) on page D5-1177.
- [D5.60 SCVTF \(scalar, fixed-point\)](#) on page D5-1178.
- [D5.61 SCVTF \(scalar, integer\)](#) on page D5-1179.
- [D5.62 SHL \(scalar\)](#) on page D5-1180.
- [D5.63 SLI \(scalar\)](#) on page D5-1181.
- [D5.64 SQABS \(scalar\)](#) on page D5-1182.
- [D5.65 SQADD \(scalar\)](#) on page D5-1183.
- [D5.66 SQDMLAL \(scalar, by element\)](#) on page D5-1184.
- [D5.67 SQDMLAL \(scalar\)](#) on page D5-1185.
- [D5.68 SQDMLSL \(scalar, by element\)](#) on page D5-1186.
- [D5.69 SQDMLSL \(scalar\)](#) on page D5-1187.
- [D5.70 SQDMULLH \(scalar, by element\)](#) on page D5-1188.
- [D5.71 SQDMULH \(scalar\)](#) on page D5-1189.
- [D5.72 SQDMULL \(scalar, by element\)](#) on page D5-1190.
- [D5.73 SQDMULL \(scalar\)](#) on page D5-1191.
- [D5.74 SQNEG \(scalar\)](#) on page D5-1192.
- [D5.75 SQRDMLAH \(scalar, by element\)](#) on page D5-1193.
- [D5.76 SQRDMLAH \(scalar\)](#) on page D5-1194.
- [D5.77 SQRDMLSH \(scalar, by element\)](#) on page D5-1195.
- [D5.78 SQRDMLSH \(scalar\)](#) on page D5-1196.
- [D5.79 SQRDMULH \(scalar, by element\)](#) on page D5-1197.

- [D5.80 SQRDMULH \(scalar\)](#) on page D5-1198.
- [D5.81 SQRSHL \(scalar\)](#) on page D5-1199.
- [D5.82 SQRSHRN \(scalar\)](#) on page D5-1200.
- [D5.83 SQRSHRUN \(scalar\)](#) on page D5-1201.
- [D5.84 SQSHL \(scalar, immediate\)](#) on page D5-1202.
- [D5.85 SQSHL \(scalar, register\)](#) on page D5-1203.
- [D5.86 SQSHLU \(scalar\)](#) on page D5-1204.
- [D5.87 SQSHRN \(scalar\)](#) on page D5-1205.
- [D5.88 SQSHRUN \(scalar\)](#) on page D5-1206.
- [D5.89 SQSUB \(scalar\)](#) on page D5-1207.
- [D5.90 SQXTN \(scalar\)](#) on page D5-1208.
- [D5.91 SQXTUN \(scalar\)](#) on page D5-1209.
- [D5.92 SRI \(scalar\)](#) on page D5-1210.
- [D5.93 SRSHL \(scalar\)](#) on page D5-1211.
- [D5.94 SRSHR \(scalar\)](#) on page D5-1212.
- [D5.95 SRSRA \(scalar\)](#) on page D5-1213.
- [D5.96 SSHL \(scalar\)](#) on page D5-1214.
- [D5.97 SSHR \(scalar\)](#) on page D5-1215.
- [D5.98 SSRA \(scalar\)](#) on page D5-1216.
- [D5.99 SUB \(scalar\)](#) on page D5-1217.
- [D5.100 SUQADD \(scalar\)](#) on page D5-1218.
- [D5.101 UCVTF \(scalar, fixed-point\)](#) on page D5-1219.
- [D5.102 UCVTF \(scalar, integer\)](#) on page D5-1220.
- [D5.103 UQADD \(scalar\)](#) on page D5-1221.
- [D5.104 UQRSHL \(scalar\)](#) on page D5-1222.
- [D5.105 UQRSHRN \(scalar\)](#) on page D5-1223.
- [D5.106 UQSHL \(scalar, immediate\)](#) on page D5-1224.
- [D5.107 UQSHL \(scalar, register\)](#) on page D5-1225.
- [D5.108 UQSHRN \(scalar\)](#) on page D5-1226.
- [D5.109 UQSUB \(scalar\)](#) on page D5-1227.
- [D5.110 UQXTN \(scalar\)](#) on page D5-1228.
- [D5.111 URSHL \(scalar\)](#) on page D5-1229.
- [D5.112 URSHR \(scalar\)](#) on page D5-1230.
- [D5.113 URSRA \(scalar\)](#) on page D5-1231.
- [D5.114 USHL \(scalar\)](#) on page D5-1232.
- [D5.115 USHR \(scalar\)](#) on page D5-1233.
- [D5.116 USQADD \(scalar\)](#) on page D5-1234.
- [D5.117 USRA \(scalar\)](#) on page D5-1235.

## D5.1 A64 SIMD scalar instructions in alphabetical order

A summary of the A64 SIMD scalar instructions that are supported.

**Table D5-1 Summary of A64 SIMD scalar instructions**

Mnemonic	Brief description	See
ABS (scalar)	Absolute value (vector)	<a href="#">D5.2 ABS (scalar) on page D5-1115</a>
ADD (scalar)	Add (vector)	<a href="#">D5.3 ADD (scalar) on page D5-1116</a>
ADDP (scalar)	Add Pair of elements (scalar)	<a href="#">D5.4 ADDP (scalar) on page D5-1117</a>
CMEQ (scalar, register)	Compare bitwise Equal (vector)	<a href="#">D5.5 CMEQ (scalar, register) on page D5-1118</a>
CMEQ (scalar, zero)	Compare bitwise Equal to zero (vector)	<a href="#">D5.6 CMEQ (scalar, zero) on page D5-1119</a>
CMGE (scalar, register)	Compare signed Greater than or Equal (vector)	<a href="#">D5.7 CMGE (scalar, register) on page D5-1120</a>
CMGE (scalar, zero)	Compare signed Greater than or Equal to zero (vector)	<a href="#">D5.8 CMGE (scalar, zero) on page D5-1121</a>
CMGT (scalar, register)	Compare signed Greater than (vector)	<a href="#">D5.9 CMGT (scalar, register) on page D5-1122</a>
CMGT (scalar, zero)	Compare signed Greater than zero (vector)	<a href="#">D5.10 CMGT (scalar, zero) on page D5-1123</a>
CMHI (scalar, register)	Compare unsigned Higher (vector)	<a href="#">D5.11 CMHI (scalar, register) on page D5-1124</a>
CMHS (scalar, register)	Compare unsigned Higher or Same (vector)	<a href="#">D5.12 CMHS (scalar, register) on page D5-1125</a>
CMLE (scalar, zero)	Compare signed Less than or Equal to zero (vector)	<a href="#">D5.13 CMLE (scalar, zero) on page D5-1126</a>
CMLT (scalar, zero)	Compare signed Less than zero (vector)	<a href="#">D5.14 CMLT (scalar, zero) on page D5-1127</a>
CMTST (scalar)	Compare bitwise Test bits nonzero (vector)	<a href="#">D5.15 CMTST (scalar) on page D5-1128</a>
DUP (scalar, element)	Duplicate vector element to scalar	<a href="#">D5.16 DUP (scalar, element) on page D5-1129</a>
FABD (scalar)	Floating-point Absolute Difference (vector)	<a href="#">D5.17 FABD (scalar) on page D5-1130</a>
FACGE (scalar)	Floating-point Absolute Compare Greater than or Equal (vector)	<a href="#">D5.18 FACGE (scalar) on page D5-1131</a>
FACGT (scalar)	Floating-point Absolute Compare Greater than (vector)	<a href="#">D5.19 FACGT (scalar) on page D5-1132</a>
FADDP (scalar)	Floating-point Add Pair of elements (scalar)	<a href="#">D5.20 FADDP (scalar) on page D5-1133</a>
FCMEQ (scalar, register)	Floating-point Compare Equal (vector)	<a href="#">D5.21 FCMEQ (scalar, register) on page D5-1134</a>
FCMEQ (scalar, zero)	Floating-point Compare Equal to zero (vector)	<a href="#">D5.22 FCMEQ (scalar, zero) on page D5-1135</a>
FCMGE (scalar, register)	Floating-point Compare Greater than or Equal (vector)	<a href="#">D5.23 FCMGE (scalar, register) on page D5-1136</a>
FCMGE (scalar, zero)	Floating-point Compare Greater than or Equal to zero (vector)	<a href="#">D5.24 FCMGE (scalar, zero) on page D5-1137</a>
FCMGT (scalar, register)	Floating-point Compare Greater than (vector)	<a href="#">D5.25 FCMGT (scalar, register) on page D5-1138</a>
FCMGT (scalar, zero)	Floating-point Compare Greater than zero (vector)	<a href="#">D5.26 FCMGT (scalar, zero) on page D5-1139</a>

**Table D5-1 Summary of A64 SIMD scalar instructions (continued)**

Mnemonic	Brief description	See
FCMLA (scalar, by element)	Floating-point Complex Multiply Accumulate (by element)	<a href="#">D5.27 FCMLA (scalar; by element) on page D5-1140</a>
FCMLE (scalar, zero)	Floating-point Compare Less than or Equal to zero (vector)	<a href="#">D5.28 FCMLE (scalar; zero) on page D5-1142</a>
FCMLT (scalar, zero)	Floating-point Compare Less than zero (vector)	<a href="#">D5.29 FCMLT (scalar; zero) on page D5-1143</a>
FCVTAS (scalar)	Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector)	<a href="#">D5.30 FCVTAS (scalar) on page D5-1144</a>
FCVTAU (scalar)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector)	<a href="#">D5.31 FCVTAU (scalar) on page D5-1145</a>
FCVTMS (scalar)	Floating-point Convert to Signed integer, rounding toward Minus infinity (vector)	<a href="#">D5.32 FCVTMS (scalar) on page D5-1146</a>
FCVTMU (scalar)	Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector)	<a href="#">D5.33 FCVTMU (scalar) on page D5-1147</a>
FCVTNS (scalar)	Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector)	<a href="#">D5.34 FCVTNS (scalar) on page D5-1148</a>
FCVTNU (scalar)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector)	<a href="#">D5.35 FCVTNU (scalar) on page D5-1149</a>
FCVTPS (scalar)	Floating-point Convert to Signed integer, rounding toward Plus infinity (vector)	<a href="#">D5.36 FCVTPS (scalar) on page D5-1150</a>
FCVTPU (scalar)	Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector)	<a href="#">D5.37 FCVTPU (scalar) on page D5-1151</a>
FCVTXN (scalar)	Floating-point Convert to lower precision Narrow, rounding to odd (vector)	<a href="#">D5.38 FCVTXN (scalar) on page D5-1152</a>
FCVTZS (scalar, fixed-point)	Floating-point Convert to Signed fixed-point, rounding toward Zero (vector)	<a href="#">D5.39 FCVTZS (scalar, fixed-point) on page D5-1153</a>
FCVTZS (scalar, integer)	Floating-point Convert to Signed integer, rounding toward Zero (vector)	<a href="#">D5.40 FCVTZS (scalar, integer) on page D5-1154</a>
FCVTZU (scalar, fixed-point)	Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector)	<a href="#">D5.41 FCVTZU (scalar; fixed-point) on page D5-1155</a>
FCVTZU (scalar, integer)	Floating-point Convert to Unsigned integer, rounding toward Zero (vector)	<a href="#">D5.42 FCVTZU (scalar, integer) on page D5-1156</a>
FMAXNMP (scalar)	Floating-point Maximum Number of Pair of elements (scalar)	<a href="#">D5.43 FMAXNMP (scalar) on page D5-1157</a>
FMAXP (scalar)	Floating-point Maximum of Pair of elements (scalar)	<a href="#">D5.44 FMAXP (scalar) on page D5-1158</a>
FMINNMP (scalar)	Floating-point Minimum Number of Pair of elements (scalar)	<a href="#">D5.45 FMINNMP (scalar) on page D5-1159</a>
FMINP (scalar)	Floating-point Minimum of Pair of elements (scalar)	<a href="#">D5.46 FMINP (scalar) on page D5-1160</a>
FMLA (scalar, by element)	Floating-point fused Multiply-Add to accumulator (by element)	<a href="#">D5.47 FMLA (scalar; by element) on page D5-1161</a>

**Table D5-1 Summary of A64 SIMD scalar instructions (continued)**

Mnemonic	Brief description	See
FMLAL, (scalar, by element)	Floating-point fused Multiply-Add Long to accumulator (by element)	<a href="#">D5.48 FMLAL, (scalar, by element) on page D5-1163</a>
FMLS (scalar, by element)	Floating-point fused Multiply-Subtract from accumulator (by element)	<a href="#">D5.49 FMLS (scalar, by element) on page D5-1164</a>
FMLSL, (scalar, by element)	Floating-point fused Multiply-Subtract Long from accumulator (by element)	<a href="#">D5.50 FMLSL, (scalar, by element) on page D5-1166</a>
FMUL (scalar, by element)	Floating-point Multiply (by element)	<a href="#">D5.51 FMUL (scalar, by element) on page D5-1167</a>
FMULX (scalar, by element)	Floating-point Multiply extended (by element)	<a href="#">D5.52 FMULX (scalar, by element) on page D5-1169</a>
FMULX (scalar)	Floating-point Multiply extended	<a href="#">D5.53 FMULX (scalar) on page D5-1171</a>
FRECPE (scalar)	Floating-point Reciprocal Estimate	<a href="#">D5.54 FRECPE (scalar) on page D5-1172</a>
FRECPS (scalar)	Floating-point Reciprocal Step	<a href="#">D5.55 FRECPS (scalar) on page D5-1173</a>
FRSQRTE (scalar)	Floating-point Reciprocal Square Root Estimate	<a href="#">D5.56 FRSQRTE (scalar) on page D5-1174</a>
FRSQRTS (scalar)	Floating-point Reciprocal Square Root Step	<a href="#">D5.57 FRSQRTS (scalar) on page D5-1175</a>
MOV (scalar)	Move vector element to scalar	<a href="#">D5.58 MOV (scalar) on page D5-1176</a>
NEG (scalar)	Negate (vector)	<a href="#">D5.59 NEG (scalar) on page D5-1177</a>
SCVTF (scalar, fixed-point)	Signed fixed-point Convert to Floating-point (vector)	<a href="#">D5.60 SCVTF (scalar, fixed-point) on page D5-1178</a>
SCVTF (scalar, integer)	Signed integer Convert to Floating-point (vector)	<a href="#">D5.61 SCVTF (scalar, integer) on page D5-1179</a>
SHL (scalar)	Shift Left (immediate)	<a href="#">D5.62 SHL (scalar) on page D5-1180</a>
SLI (scalar)	Shift Left and Insert (immediate)	<a href="#">D5.63 SLI (scalar) on page D5-1181</a>
SQABS (scalar)	Signed saturating Absolute value	<a href="#">D5.64 SQABS (scalar) on page D5-1182</a>
SQADD (scalar)	Signed saturating Add	<a href="#">D5.65 SQADD (scalar) on page D5-1183</a>
SQDMLAL (scalar, by element)	Signed saturating Doubling Multiply-Add Long (by element)	<a href="#">D5.66 SQDMLAL (scalar, by element) on page D5-1184</a>
SQDMLAL (scalar)	Signed saturating Doubling Multiply-Add Long	<a href="#">D5.67 SQDMLAL (scalar) on page D5-1185</a>
SQDMLSL (scalar, by element)	Signed saturating Doubling Multiply-Subtract Long (by element)	<a href="#">D5.68 SQDMLSL (scalar, by element) on page D5-1186</a>
SQDMLSL (scalar)	Signed saturating Doubling Multiply-Subtract Long	<a href="#">D5.69 SQDMLSL (scalar) on page D5-1187</a>
SQDMULH (scalar, by element)	Signed saturating Doubling Multiply returning High half (by element)	<a href="#">D5.70 SQDMULH (scalar, by element) on page D5-1188</a>
SQDMULH (scalar)	Signed saturating Doubling Multiply returning High half	<a href="#">D5.71 SQDMULH (scalar) on page D5-1189</a>
SQDMULL (scalar, by element)	Signed saturating Doubling Multiply Long (by element)	<a href="#">D5.72 SQDMULL (scalar, by element) on page D5-1190</a>

**Table D5-1 Summary of A64 SIMD scalar instructions (continued)**

Mnemonic	Brief description	See
SQDMULL (scalar)	Signed saturating Doubling Multiply Long	<a href="#">D5.73 SQDMULL (scalar) on page D5-1191</a>
SQNEG (scalar)	Signed saturating Negate	<a href="#">D5.74 SQNEG (scalar) on page D5-1192</a>
SQRDMLAH (scalar, by element)	Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element)	<a href="#">D5.75 SQRDMLAH (scalar, by element) on page D5-1193</a>
SQRDMLAH (scalar)	Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector)	<a href="#">D5.76 SQRDMLAH (scalar) on page D5-1194</a>
SQRDMLSH (scalar, by element)	Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element)	<a href="#">D5.77 SQRDMLSH (scalar, by element) on page D5-1195</a>
SQRDMLSH (scalar)	Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector)	<a href="#">D5.78 SQRDMLSH (scalar) on page D5-1196</a>
SQRDMULH (scalar, by element)	Signed saturating Rounding Doubling Multiply returning High half (by element)	<a href="#">D5.79 SQRDMULH (scalar, by element) on page D5-1197</a>
SQRDMULH (scalar)	Signed saturating Rounding Doubling Multiply returning High half	<a href="#">D5.80 SQRDMULH (scalar) on page D5-1198</a>
SQRSHL (scalar)	Signed saturating Rounding Shift Left (register)	<a href="#">D5.81 SQRSHL (scalar) on page D5-1199</a>
SQRSHRN (scalar)	Signed saturating Rounded Shift Right Narrow (immediate)	<a href="#">D5.82 SQRSHRN (scalar) on page D5-1200</a>
SQRSHRUN (scalar)	Signed saturating Rounded Shift Right Unsigned Narrow (immediate)	<a href="#">D5.83 SQRSHRUN (scalar) on page D5-1201</a>
SQSHL (scalar, immediate)	Signed saturating Shift Left (immediate)	<a href="#">D5.84 SQSHL (scalar, immediate) on page D5-1202</a>
SQSHL (scalar, register)	Signed saturating Shift Left (register)	<a href="#">D5.85 SQSHL (scalar, register) on page D5-1203</a>
SQSHLU (scalar)	Signed saturating Shift Left Unsigned (immediate)	<a href="#">D5.86 SQSHLU (scalar) on page D5-1204</a>
SQSHRN (scalar)	Signed saturating Shift Right Narrow (immediate)	<a href="#">D5.87 SQSHRN (scalar) on page D5-1205</a>
SQSHRUN (scalar)	Signed saturating Shift Right Unsigned Narrow (immediate)	<a href="#">D5.88 SQSHRUN (scalar) on page D5-1206</a>
SQSUB (scalar)	Signed saturating Subtract	<a href="#">D5.89 SQSUB (scalar) on page D5-1207</a>
SQXTN (scalar)	Signed saturating extract Narrow	<a href="#">D5.90 SQXTN (scalar) on page D5-1208</a>
SQXTUN (scalar)	Signed saturating extract Unsigned Narrow	<a href="#">D5.91 SQXTUN (scalar) on page D5-1209</a>
SRI (scalar)	Shift Right and Insert (immediate)	<a href="#">D5.92 SRI (scalar) on page D5-1210</a>
SRSHL (scalar)	Signed Rounding Shift Left (register)	<a href="#">D5.93 SRSHL (scalar) on page D5-1211</a>
SRSHR (scalar)	Signed Rounding Shift Right (immediate)	<a href="#">D5.94 SRSR (scalar) on page D5-1212</a>
SRSRA (scalar)	Signed Rounding Shift Right and Accumulate (immediate)	<a href="#">D5.95 SRSRA (scalar) on page D5-1213</a>

**Table D5-1 Summary of A64 SIMD scalar instructions (continued)**

Mnemonic	Brief description	See
SSH <sub>L</sub> (scalar)	Signed Shift Left (register)	<a href="#">D5.96 SSHL (scalar) on page D5-1214</a>
SSH <sub>R</sub> (scalar)	Signed Shift Right (immediate)	<a href="#">D5.97 SSHR (scalar) on page D5-1215</a>
SSRA (scalar)	Signed Shift Right and Accumulate (immediate)	<a href="#">D5.98 SSRA (scalar) on page D5-1216</a>
SUB (scalar)	Subtract (vector)	<a href="#">D5.99 SUB (scalar) on page D5-1217</a>
SUQADD (scalar)	Signed saturating Accumulate of Unsigned value	<a href="#">D5.100 SUQADD (scalar) on page D5-1218</a>
UCVTF (scalar, fixed-point)	Unsigned fixed-point Convert to Floating-point (vector)	<a href="#">D5.101 UCVTF (scalar; fixed-point) on page D5-1219</a>
UCVTF (scalar, integer)	Unsigned integer Convert to Floating-point (vector)	<a href="#">D5.102 UCVTF (scalar, integer) on page D5-1220</a>
UQADD (scalar)	Unsigned saturating Add	<a href="#">D5.103 UQADD (scalar) on page D5-1221</a>
UQRSHL (scalar)	Unsigned saturating Rounding Shift Left (register)	<a href="#">D5.104 UQRSHL (scalar) on page D5-1222</a>
UQRSHRN (scalar)	Unsigned saturating Rounded Shift Right Narrow (immediate)	<a href="#">D5.105 UQRSHRN (scalar) on page D5-1223</a>
UQS <sub>L</sub> (scalar, immediate)	Unsigned saturating Shift Left (immediate)	<a href="#">D5.106 UQS<sub>L</sub> (scalar, immediate) on page D5-1224</a>
UQS <sub>H</sub> (scalar, register)	Unsigned saturating Shift Left (register)	<a href="#">D5.107 UQS<sub>H</sub> (scalar, register) on page D5-1225</a>
UQSHRN (scalar)	Unsigned saturating Shift Right Narrow (immediate)	<a href="#">D5.108 UQSHRN (scalar) on page D5-1226</a>
UQS <sub>B</sub> (scalar)	Unsigned saturating Subtract	<a href="#">D5.109 UQS<sub>B</sub> (scalar) on page D5-1227</a>
UQXTN (scalar)	Unsigned saturating extract Narrow	<a href="#">D5.110 UQXTN (scalar) on page D5-1228</a>
URSHL (scalar)	Unsigned Rounding Shift Left (register)	<a href="#">D5.111 URSHL (scalar) on page D5-1229</a>
URSHR (scalar)	Unsigned Rounding Shift Right (immediate)	<a href="#">D5.112 URSHR (scalar) on page D5-1230</a>
URSRA (scalar)	Unsigned Rounding Shift Right and Accumulate (immediate)	<a href="#">D5.113 URSRA (scalar) on page D5-1231</a>
USHL (scalar)	Unsigned Shift Left (register)	<a href="#">D5.114 USHL (scalar) on page D5-1232</a>
USHR (scalar)	Unsigned Shift Right (immediate)	<a href="#">D5.115 USHR (scalar) on page D5-1233</a>
USQADD (scalar)	Unsigned saturating Accumulate of Signed value	<a href="#">D5.116 USQADD (scalar) on page D5-1234</a>
USRA (scalar)	Unsigned Shift Right and Accumulate (immediate)	<a href="#">D5.117 USRA (scalar) on page D5-1235</a>

## D5.2 ABS (scalar)

Absolute value (vector).

### Syntax

ABS  $Vd$ ,  $Vn$

Where:

$V$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the SIMD and FP source register.

### Usage

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD and FP register, puts the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.3 ADD (scalar)

Add (vector).

### Syntax

ADD  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Add (vector). This instruction adds corresponding elements in the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.4 ADDP (scalar)

Add Pair of elements (scalar).

### Syntax

ADDP  $Vd, Vn.T$

Where:

$V$

Is the destination width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$Vn$

Is the name of the SIMD and FP source register.

$T$

Is the source arrangement specifier, 2D.

### Usage

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD and FP register and writes the scalar result into the destination SIMD and FP register.

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

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.5 CMEQ (scalar, register)

Compare bitwise Equal (vector).

### Syntax

CMEQ  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD and FP register with the corresponding vector element from the second source SIMD and FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.6 CMEQ (scalar, zero)

Compare bitwise Equal to zero (vector).

### Syntax

CMEQ  $Vd$ ,  $Vn$ , #0

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the SIMD and FP source register.

### Usage

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.7 CMGE (scalar, register)

Compare signed Greater than or Equal (vector).

### Syntax

CMGE  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.8 CMGE (scalar, zero)

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

### Syntax

CMGE  $Vd$ ,  $Vn$ , #0

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the SIMD and FP source register.

### Usage

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.9 CMGT (scalar, register)

Compare signed Greater than (vector).

### Syntax

CMGT  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.10 CMGT (scalar, zero)

Compare signed Greater than zero (vector).

### Syntax

CMGT  $Vd$ ,  $Vn$ , #0

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the SIMD and FP source register.

### Usage

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.11 CMHI (scalar, register)

Compare unsigned Higher (vector).

### Syntax

CMHI  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.12 CMHS (scalar, register)

Compare unsigned Higher or Same (vector).

### Syntax

CMHS  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.13 CMLE (scalar, zero)

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

### Syntax

CMLE  $Vd$ ,  $Vn$ , #0

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the SIMD and FP source register.

### Usage

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.14 CMLT (scalar, zero)

Compare signed Less than zero (vector).

### Syntax

CMLT *Vd*, *Vn*, #0

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Usage

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

## D5.15 CMTST (scalar)

Compare bitwise Test bits nonzero (vector).

### Syntax

CMTST *Vd, Vn,Vm*

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Usage

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD and FP register, performs an AND with the corresponding vector element in the second source SIMD and FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

## D5.16 DUP (scalar, element)

Duplicate vector element to scalar.

This instruction is used by the alias MOV (scalar).

### Syntax

DUP  $Vd, Vn.T[index]$

Where:

$V$

Is the destination width specifier, and can be one of the values shown in Usage.

$d$

Is the number of the SIMD and FP destination register.

$T$

Is the element width specifier, and can be one of the values shown in Usage.

$Vn$

Is the name of the SIMD and FP source register.

$index$

Is the element index, in the range shown in Usage.

### Usage

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD and FP register into a scalar or each element in a vector, and writes the result to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D5-2 DUP (Scalar) specifier combinations**

$V$	$T$	$index$
B	B	0 to 15
H	H	0 to 7
S	S	0 to 3
D	D	0 or 1

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.17 FABD (scalar)

Floating-point Absolute Difference (vector).

### Syntax

FABD *Hd*, *Hn*, *Hm* ; Scalar half precision

FABD *Vd*, *Vn*, *Vm* ; Scalar single-precision and double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD and FP register, from the corresponding floating-point values in the elements of the first source SIMD and FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.18 FACGE (scalar)

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

### Syntax

```
FACGE Hd, Hn, Hm ; Scalar half precision  
FACGE Vd, Vn,Vm ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD and FP register with the absolute value of the corresponding floating-point value in the second source SIMD and FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.19 FACGT (scalar)

Floating-point Absolute Compare Greater than (vector).

### Syntax

`FACGT Hd, Hn, Hm ; Scalar half precision`

`FACGT Vd, Vn,Vm ; Scalar single-precision and double-precision`

Where:

`Hd`

Is the 16-bit name of the SIMD and FP destination register.

`Hn`

Is the 16-bit name of the first SIMD and FP source register.

`Hm`

Is the 16-bit name of the second SIMD and FP source register.

`V`

Is a width specifier, and can be either S or D.

`d`

Is the number of the SIMD and FP destination register.

`n`

Is the number of the first SIMD and FP source register.

`m`

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD and FP register with the absolute value of the corresponding vector element in the second source SIMD and FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.20 FADDP (scalar)

Floating-point Add Pair of elements (scalar).

### Syntax

FADDP *Vd*, *Vn.T* ; Half-precision  
FADDP *Vd*, *Vn.T* ; Single-precision and double-precision

Where:

**v**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Can be one of S or D.

**T**

Is the source arrangement specifier:

#### Half-precision

Must be 2H.

#### Single-precision and double-precision

Can be one of 2S or 2D.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD and FP register and writes the scalar result into the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.21 FCMEQ (scalar, register)

Floating-point Compare Equal (vector).

### Syntax

```
FCMEQ Hd, Hn, Hm ; Scalar half precision  
FCMEQ Vd, Vn,Vm ; Scalar single-precision and double-precision
```

Where:

**Hd**

Is the 16-bit name of the SIMD and FP destination register.

**Hn**

Is the 16-bit name of the first SIMD and FP source register.

**Hm**

Is the 16-bit name of the second SIMD and FP source register.

**V**

Is a width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**m**

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD and FP register, with the corresponding floating-point value from the second source SIMD and FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.22 FCMEQ (scalar, zero)

Floating-point Compare Equal to zero (vector).

### Syntax

```
FCMEQ Hd, Hn, #0.0 ; Scalar half precision  
FCMEQ Vd, Vn, #0.0 ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.23 FCMGE (scalar, register)

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

### Syntax

FCMGE *Hd*, *Hn*, *Hm* ; Scalar half precision

FCMGE *Vd*, *Vn*, *Vm* ; Scalar single-precision and double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD and FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD and FP register sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.24 FCMGE (scalar, zero)

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

### Syntax

FCMGE *Hd*, *Hn*, #0.0 ; Scalar half precision

FCMGE *Vd*, *Vn*, #0.0 ; Scalar single-precision and double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.25 FCMGT (scalar, register)

Floating-point Compare Greater than (vector).

### Syntax

FCMGT *Hd*, *Hn*, *Hm* ; Scalar half precision

FCMGT *Vd*, *Vn*, *Vm* ; Scalar single-precision and double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD and FP register and if the value is greater than the corresponding floating-point value in the second source SIMD and FP register sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.26 FCMGT (scalar, zero)

Floating-point Compare Greater than zero (vector).

### Syntax

FCMGT *Hd*, *Hn*, #0.0 ; Scalar half precision

FCMGT *Vd*, *Vn*, #0.0 ; Scalar single-precision and double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.27 FCMLA (scalar, by element)

Floating-point Complex Multiply Accumulate (by element).

### Syntax

FCMLA *Vd.T, Vn.T, Vm.Ts[index], #rotate*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register in the range 0 to 31.

*Ts*

Is an element size specifier, and can be either H or S.

*index*

Is the element index, in the range shown in Usage.

*rotate*

Is the rotation, and can be one of the values shown in Usage.

### Architectures supported (scalar)

Supported in the Armv8.3-A architecture and later.

### Usage

This instruction multiplies the two source complex numbers from the *Vm* and the *Vn* vector registers and adds the result to the corresponding complex number in the destination *Vd* vector register. The number of complex numbers that can be stored in the *Vm*, the *Vn*, and the *Vd* registers is calculated as the vector register size divided by the length of each complex number. These lengths are 16 for half-precision, 32 for single-precision, and 64 for double-precision. Each complex number is represented in a SIMD&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

None, one, or both of the two vector elements that are read from each of the numbers in the *Vm* source SIMD and FP register can be negated based on the rotation value:

- If the rotation is 0, none of the vector elements are negated.
- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 180, both vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

The indexed element variant of this instruction is available for half-precision and single-precision number values. For this variant, the index value determines the position in the *Vm* source vector register of the single source value that is used to multiply each of the complex numbers in the *Vn* source vector register. The index value is encoded as H:L for half-precision values, or H for single-precision 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-3 FCMLA (Scalar) specifier combinations**

<i>T</i>	<i>Ts</i>
4H	H
8H	H
4S	S

***Related reference***

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

***Related information***

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.28 FCMLE (scalar, zero)

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

### Syntax

FCMLE *Hd*, *Hn*, #0.0 ; Scalar half precision

FCMLE *Vd*, *Vn*, #0.0 ; Scalar single-precision and double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.29 FCMLT (scalar, zero)

Floating-point Compare Less than zero (vector).

### Syntax

```
FCMLT Hd, Hn, #0.0 ; Scalar half precision  
FCMLT Vd, Vn, #0.0 ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.30 FCVTAS (scalar)

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

### Syntax

```
FCVTAS Hd, Hn ; Scalar half precision  
FCVTAS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.31 FCVTAU (scalar)

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

### Syntax

```
FCVTAU Hd, Hn ; Scalar half precision  
FCVTAU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.32 FCVTMS (scalar)

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

### Syntax

```
FCVTMS Hd, Hn ; Scalar half precision  
FCVTMS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.33 FCVTMU (scalar)

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

### Syntax

```
FCVTMU Hd, Hn ; Scalar half precision  
FCVTMU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

**Hd**

Is the 16-bit name of the SIMD and FP destination register.

**Hn**

Is the 16-bit name of the SIMD and FP source register.

**V**

Is a width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.34 FCVTNS (scalar)

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

### Syntax

```
FCVTNS Hd, Hn ; Scalar half precision  
FCVTNS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.35 FCVTNU (scalar)

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

### Syntax

```
FCVTNU Hd, Hn ; Scalar half precision  
FCVTNU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

#### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

#### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.36 FCVTPS (scalar)

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

### Syntax

```
FCVTPS Hd, Hn ; Scalar half precision  
FCVTPS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

**Hd**

Is the 16-bit name of the SIMD and FP destination register.

**Hn**

Is the 16-bit name of the SIMD and FP source register.

**V**

Is a width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

#### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

#### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.37 FCVTPU (scalar)

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

### Syntax

```
FCVTPU Hd, Hn ; Scalar half precision  
FCVTPU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

#### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

#### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.38 FCVTXN (scalar)

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

### Syntax

FCVTXN *Vbd*, *Van*

Where:

*Vb*

Is the destination width specifier, S.

*d*

Is the number of the SIMD and FP destination register.

*Va*

Is the source width specifier, D.

*n*

Is the number of the SIMD and FP source register.

### Usage

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD and FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD and FP register.

#### Note

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.39 FCVTZS (scalar, fixed-point)

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

### Syntax

FCVTZS *Vd*, *Vn*, #*fbits*

Where:

*v*

Is a width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*fbits*

Is the number of fractional bits, in the range 1 to the operand width.

### Usage

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-4 FCVTZS (Scalar) specifier combinations**

<i>V</i>	<i>fbits</i>
H	
S	1 to 32
D	1 to 64

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.40 FCVTZS (scalar, integer)

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

### Syntax

```
FCVTZS Hd, Hn ; Scalar half precision  
FCVTZS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.41 FCVTZU (scalar, fixed-point)

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

### Syntax

`FCVTZU Vd, Vn, #fbits`

Where:

**v**

Is a width specifier, and can be one of the values shown in Usage.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**fbits**

Is the number of fractional bits, in the range 1 to the operand width.

### Usage

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-5 FCVTZU (Scalar) specifier combinations**

V	fbits
H	
S	1 to 32
D	1 to 64

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.42 FCVTZU (scalar, integer)

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

### Syntax

```
FCVTZU Hd, Hn ; Scalar half precision  
FCVTZU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.43 FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar).

### Syntax

FMAXNMP *Vd*, *Vn.T* ; Half-precision  
FMAXNMP *Vd*, *Vn.T* ; Single-precision and double-precision

Where:

**v**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Can be one of S or D.

**T**

Is the source arrangement specifier:

#### Half-precision

Must be 2H.

#### Single-precision and double-precision

Can be one of 2S or 2D.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD and FP register and writes the largest of the floating-point values as a scalar to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.44 FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar).

### Syntax

FMAXP *Vd*, *Vn.T* ; Half-precision  
FMAXP *Vd*, *Vn.T* ; Single-precision and double-precision

Where:

**v**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Can be one of S or D.

**T**

Is the source arrangement specifier:

#### Half-precision

Must be 2H.

#### Single-precision and double-precision

Can be one of 2S or 2D.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD and FP register and writes the largest of the floating-point values as a scalar to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.45 FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar).

### Syntax

```
FMINNMP Vd, Vn.T ; Half-precision  
FMINNMP Vd, Vn.T ; Single-precision and double-precision
```

Where:

**V**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Can be one of S or D.

**T**

Is the source arrangement specifier:

#### Half-precision

Must be 2H.

#### Single-precision and double-precision

Can be one of 2S or 2D.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD and FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.46 FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar).

### Syntax

FMINP *Vd*, *Vn.T* ; Half-precision  
FMINP *Vd*, *Vn.T* ; Single-precision and double-precision

Where:

**v**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Can be one of S or D.

**T**

Is the source arrangement specifier:

#### Half-precision

Must be 2H.

#### Single-precision and double-precision

Can be one of 2S or 2D.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD and FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.47 FMLA (scalar, by element)

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

### Syntax

FMLA *Hd*, *Hn*, *Vm.H*[*index*] ; Scalar, half-precision

FMLA *Vd*, *Vn*, *Vm.Ts*[*index*] ; Scalar, single-precision and double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Vm*

The value depends on the instruction variant:

#### Scalar, half-precision

For the half-precision variant: is the name of the second SIMD and FP source register, in the range V0 to V15

#### Scalar, single-precision and double-precision

For the single-precision and double-precision variant: is the name of the second SIMD and FP source register in the range 0 to 31.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*Ts*

Is an element size specifier, and can be either S or D.

*index*

Is the element index, in the range 0 to 7.

For the single-precision and double-precision variant: is the element index H:L, H.

### Architectures supported (scalar)

Supported in Armv8.2 and later.

### Usage

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the results in the vector elements of the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-6 FMLA (Scalar, single-precision and double-precision) specifier combinations**

<i>V</i>	<i>Ts</i>	<i>index</i>
S	S	0 to 3
D	D	0 or 1

**Related reference**

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D5.48 FMLAL, (scalar, by element)

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

### Syntax

```
FMLAL Vd.Ta, Vn.Tb, Vm.H[index] ; FMLAL  
FMLAL2 Vd.Ta, Vn.Tb, Vm.H[index] ; FMLAL2
```

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of 2S or 4S.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of 2H or 4H.

**Vm**

Is the name of the second SIMD and FP source register.

**index**

Is the element index.

### Architectures supported (scalar)

Supported in Armv8.2 and later.

### Usage

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the product to the corresponding vector element of the destination SIMD and FP register. The instruction does not round the result of the multiply before the accumulation.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

---

#### Note

ID\_AA64ISAR0\_EL1.FHM indicates whether this instruction is supported. See ID\_AA64ISAR0\_EL1 in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.49 FMLS (scalar, by element)

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

### Syntax

`FMLS Hd, Hn, Vm.H[index] ; Scalar, half-precision`

`FMLS Vd, Vn, Vm.Ts[index] ; Scalar, single-precision and double-precision`

Where:

**Hd**

Is the 16-bit name of the SIMD and FP destination register.

**Hn**

Is the 16-bit name of the first SIMD and FP source register.

**Vm**

The value depends on the instruction variant:

#### Scalar, half-precision

For the half-precision variant: is the name of the second SIMD and FP source register, in the range V0 to V15

#### Scalar, single-precision and double-precision

For the single-precision and double-precision variant: is the name of the second SIMD and FP source register in the range 0 to 31.

**V**

Is a width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**Ts**

Is an element size specifier, and can be either S or D.

**index**

Is the element index, in the range 0 to 7.

For the single-precision and double-precision variant: is the element index H:L, H.

### Architectures supported (scalar)

Supported in Armv8.2 and later.

### Usage

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and subtracts the results from the vector elements of the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-7 FMLS (Scalar, single-precision and double-precision) specifier combinations**

<i>V</i>	<i>Ts</i>	<i>index</i>
S	S	0 to 3
D	D	0 or 1

**Related reference**

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

**Related information**

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.50 FMLSL, (scalar, by element)

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

### Syntax

FMLSL *Vd.Ta, Vn.Tb, Vm.H[index]* ; FMLSL

FMLSL2 *Vd.Ta, Vn.Tb, Vm.H[index]* ; FMLSL2

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be one of 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be one of 2H or 4H.

*Vm*

Is the name of the second SIMD and FP source register.

*index*

Is the element index.

### Architectures supported (scalar)

Supported in Armv8.2 and later.

### Usage

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the product to the corresponding vector element of the destination SIMD and FP register. The instruction does not round the result of the multiply before the accumulation.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

---

#### Note

ID\_AA64ISAR0\_EL1.FHM indicates whether this instruction is supported. See ID\_AA64ISAR0\_EL1 in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.51 FMUL (scalar, by element)

Floating-point Multiply (by element).

### Syntax

```
FMUL Hd, Hn, Vm.H[index] ; Scalar, half-precision  
FMUL Vd, Vn, Vm.Ts[index] ; Scalar, single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Vm*

The value depends on the instruction variant:

#### Scalar, half-precision

For the half-precision variant: is the name of the second SIMD and FP source register, in the range V0 to V15

#### Scalar, single-precision and double-precision

For the single-precision and double-precision variant: is the name of the second SIMD and FP source register in the range 0 to 31.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*Ts*

Is an element size specifier, and can be either S or D.

*index*

Is the element index, in the range 0 to 7.

For the single-precision and double-precision variant: is the element index H:L, H.

### Architectures supported (scalar)

Supported in Armv8.2 and later.

### Usage

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-8 FMUL (Scalar, single-precision and double-precision) specifier combinations**

<i>V</i>	<i>Ts</i>	<i>index</i>
S	S	0 to 3
D	D	0 or 1

**Related reference**

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D5.52 FMULX (scalar, by element)

Floating-point Multiply extended (by element).

### Syntax

```
FMULX Hd, Hn, Vm.H[index] ; Scalar, half-precision  
FMULX Vd, Vn, Vm.Ts[index] ; Scalar, single-precision and double-precision
```

Where:

**Hd**

Is the 16-bit name of the SIMD and FP destination register.

**Hn**

Is the 16-bit name of the first SIMD and FP source register.

**Vm**

The value depends on the instruction variant:

#### Scalar, half-precision

For the half-precision variant: is the name of the second SIMD and FP source register, in the range V0 to V15

#### Scalar, single-precision and double-precision

For the single-precision and double-precision variant: is the name of the second SIMD and FP source register in the range 0 to 31.

**V**

Is a width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**Ts**

Is an element size specifier, and can be either S or D.

**index**

Is the element index, in the range 0 to 7.

For the single-precision and double-precision variant: is the element index H:L, H.

### Architectures supported (scalar)

Supported in Armv8.2 and later.

### Usage

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD and FP register by the specified floating-point value in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-9 FMULX (Scalar, single-precision and double-precision) specifier combinations**

<i>V</i>	<i>Ts</i>	<i>index</i>
S	S	0 to 3
D	D	0 or 1

**Related reference**

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

**Related information**

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.53 FMULX (scalar)

Floating-point Multiply extended.

### Syntax

```
FMULX Hd, Hn, Hm ; Scalar half precision  
FMULX Vd, Vn,Vm ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD and FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD and FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.54 FRECPE (scalar)

Floating-point Reciprocal Estimate.

### Syntax

```
FRECPE Hd, Hn ; Scalar half precision  
FRECPE Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.55 FRECPS (scalar)

Floating-point Reciprocal Step.

### Syntax

`FRECPS Hd, Hn, Hm ; Scalar half precision`

`FRECPS Vd, Vn, Vm ; Scalar single-precision and double-precision`

Where:

`Hd`

Is the 16-bit name of the SIMD and FP destination register.

`Hn`

Is the 16-bit name of the first SIMD and FP source register.

`Hm`

Is the 16-bit name of the second SIMD and FP source register.

`V`

Is a width specifier, and can be either S or D.

`d`

Is the number of the SIMD and FP destination register.

`n`

Is the number of the first SIMD and FP source register.

`m`

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD and FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.56 FRSQRTE (scalar)

Floating-point Reciprocal Square Root Estimate.

### Syntax

```
FRSQRTE Hd, Hn ; Scalar half precision  
FRSQRTE Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.57 FRSQRTS (scalar)

Floating-point Reciprocal Square Root Step.

### Syntax

FRSQRTS *Hd*, *Hn*, *Hm* ; Scalar half precision

FRSQRTS *Vd*, *Vn*, *Vm* ; Scalar single-precision and double-precision

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the first SIMD and FP source register.

*Hm*

Is the 16-bit name of the second SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD and FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.58 MOV (scalar)

Move vector element to scalar.

This instruction is an alias of DUP (element).

The equivalent instruction is DUP  $Vd$ ,  $Vn.T[index]$ .

### Syntax

MOV  $Vd$ ,  $Vn.T[index]$

Where:

$v$

Is the destination width specifier, and can be one of the values shown in Usage.

$d$

Is the number of the SIMD and FP destination register.

$Vn$

Is the name of the SIMD and FP source register.

$T$

Is the element width specifier, and can be one of the values shown in Usage.

$index$

Is the element index, in the range shown in Usage.

### Usage

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD and FP source register into a scalar, and writes the result to the SIMD and 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.

The following table shows the valid specifier combinations:

Table D5-10 MOV (Scalar) specifier combinations

$v$	$T$	$index$
B	B	0 to 15
H	H	0 to 7
S	S	0 to 3
D	D	0 or 1

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.59 NEG (scalar)

Negate (vector).

### Syntax

NEG  $Vd$ ,  $Vn$

Where:

$V$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the SIMD and FP source register.

### Usage

Negate (vector). This instruction reads each vector element from the source SIMD and FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.60 SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (vector).

### Syntax

`SCVTF Vd, Vn, #fbits`

Where:

**V**

Is a width specifier, and can be one of the values shown in Usage.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**fbits**

Is the number of fractional bits, in the range 1 to the operand width.

### Usage

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-11 SCVTF (Scalar) specifier combinations**

<b>V</b>	<b>fbits</b>
H	
S	1 to 32
D	1 to 64

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.61 SCVTF (scalar, integer)

Signed integer Convert to Floating-point (vector).

### Syntax

```
SCVTF Hd, Hn ; Scalar half precision  
SCVTF Vd, Vn ; Scalar single-precision and double-precision
```

Where:

*Hd*

Is the 16-bit name of the SIMD and FP destination register.

*Hn*

Is the 16-bit name of the SIMD and FP source register.

*V*

Is a width specifier, and can be either S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.62 SHL (scalar)

Shift Left (immediate).

### Syntax

SHL *Vd*, *Vn*, #*shift*

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the left shift amount, in the range 0 to 63.

### Usage

Shift Left (immediate). This instruction reads each value from a vector, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and 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.

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

## D5.63 SLI (scalar)

Shift Left and Insert (immediate).

### Syntax

`SLI Vd, Vn, #shift`

Where:

**v**

Is a width specifier, D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**shift**

Is the left shift amount, in the range 0 to 63.

### Usage

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD and FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD and FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.64 SQABS (scalar)

Signed saturating Absolute value.

### Syntax

SQABS  $Vd, Vn$

Where:

$v$

Is a width specifier, and can be one of B, H, S or D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the SIMD and FP source register.

### Usage

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD and FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.65 SQADD (scalar)

Signed saturating Add.

### Syntax

SQADD  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, and can be one of B, H, S or D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.66 SQMLAL (scalar, by element)

Signed saturating Doubling Multiply-Add Long (by element).

### Syntax

`SQMLAL Vad, Vbn, Vm.Ts[index]`

Where:

**Va**

Is the destination width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**Vb**

Is the source width specifier, and can be either H or S.

**n**

Is the number of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If  $Ts$  is H, then  $Vm$  must be in the range V0 to V15.
- If  $Ts$  is S, then  $Vm$  must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

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 and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD and 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 SQMLAL instruction extracts vector elements from the lower half of the first source register, while the SQMLAL2 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.

The following table shows the valid specifier combinations:

Table D5-12 SQMLAL (Scalar) specifier combinations

Va	Vb	Ts	index
S	H	H	0 to 7
D	S	S	0 to 3

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.67 SQDMLAL (scalar)

Signed saturating Doubling Multiply-Add Long.

### Syntax

`SQDMLAL Vad, Vbn, Vbm`

Where:

**Va**

Is the destination width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**Vb**

Is the source width specifier, and can be either H or S.

**n**

Is the number of the first SIMD and FP source register.

**m**

Is the number of the second SIMD and FP source register.

### Usage

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 and FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D5-13 SQDMLAL (Scalar) specifier combinations**

Va	Vb
S	H
D	S

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.68 SQDMLSL (scalar, by element)

Signed saturating Doubling Multiply-Subtract Long (by element).

### Syntax

`SQDMLSL Vad, Vbn, Vm.Ts[index]`

Where:

**Va**

Is the destination width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**Vb**

Is the source width specifier, and can be either H or S.

**n**

Is the number of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If  $Ts$  is H, then  $Vm$  must be in the range V0 to V15.
- If  $Ts$  is S, then  $Vm$  must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

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 and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D5-14 SQDMLSL (Scalar) specifier combinations**

Va	Vb	Ts	index
S	H	H	0 to 7
D	S	S	0 to 3

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.69 SQDMLSL (scalar)

Signed saturating Doubling Multiply-Subtract Long.

### Syntax

`SQDMLSL Vad, Vbn, Vbm`

Where:

**Va**

Is the destination width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**Vb**

Is the source width specifier, and can be either H or S.

**n**

Is the number of the first SIMD and FP source register.

**m**

Is the number of the second SIMD and FP source register.

### Usage

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 and FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D5-15 SQDMLSL (Scalar) specifier combinations**

Va	Vb
S	H
D	S

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.70 SQDMULH (scalar, by element)

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

### Syntax

`SQDMULH Vd, Vn, Vm.Ts[index]`

Where:

**V**

Is a width specifier, and can be either H or S.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see [D5.79 SQRDMLH \(scalar, by element\) on page D5-1197](#).

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.

The following table shows the valid specifier combinations:

**Table D5-16 SQDMULH (Scalar) specifier combinations**

<b>V</b>	<b>Ts</b>	<b>index</b>
H	H	0 to 7
S	S	0 to 3

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.71 SQDMULH (scalar)

Signed saturating Doubling Multiply returning High half.

### Syntax

SQDMULH  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, and can be either H or S.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD and FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see [D5.80 SQRDMULH \(scalar\) on page D5-1198](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.72 SQDMULL (scalar, by element)

Signed saturating Doubling Multiply Long (by element).

### Syntax

`SQDMULL Vad, Vbn, Vm.Ts[index]`

Where:

**Va**

Is the destination width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**Vb**

Is the source width specifier, and can be either H or S.

**n**

Is the number of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If  $Ts$  is H, then  $Vm$  must be in the range V0 to V15.
- If  $Ts$  is S, then  $Vm$  must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD and 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\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

**Table D5-17 SQDMULL (Scalar) specifier combinations**

Va	Vb	Ts	index
S	H	H	0 to 7
D	S	S	0 to 3

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.73 SQDMULL (scalar)

Signed saturating Doubling Multiply Long.

### Syntax

`SQDMULL Vad, Vbn, Vbm`

Where:

**Va**

Is the destination width specifier, and can be either S or D.

**d**

Is the number of the SIMD and FP destination register.

**Vb**

Is the source width specifier, and can be either H or S.

**n**

Is the number of the first SIMD and FP source register.

**m**

Is the number of the second SIMD and FP source register.

### Usage

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD and FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D5-18 SQDMULL (Scalar) specifier combinations**

Va	Vb
S	H
D	S

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.74 SQNEG (scalar)

Signed saturating Negate.

### Syntax

SQNEG *Vd, Vn*

Where:

*v*

Is a width specifier, and can be one of B, H, S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Usage

Signed saturating Negate. This instruction reads each vector element from the source SIMD and FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

## D5.75 SQRDMLAH (scalar, by element)

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

### Syntax

`SQRDMLAH Vd, Vn, Vm.Ts[index]`

Where:

**V**

Is a width specifier, and can be either H or S.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Architectures supported (scalar)

Supported in the Armv8.1 architecture and later.

### Usage

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD and FP register with the value of a vector element of the second source SIMD and FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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.

The following table shows the valid specifier combinations:

**Table D5-19 SQRDMLAH (Scalar) specifier combinations**

V	Ts	index
H	H	0 to 7
S	S	0 to 3

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.76 SQRDMLAH (scalar)

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

### Syntax

SQRDMLAH *Vd*, *Vn*, *Vm*

Where:

*v*

Is a width specifier, and can be either H or S.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.1 architecture and later.

### Usage

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD and FP register with the corresponding vector elements of the second source SIMD and FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.77 SQRDMLSH (scalar, by element)

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

### Syntax

`SQRDMLSH Vd, Vn, Vm.Ts[index]`

Where:

**V**

Is a width specifier, and can be either H or S.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Architectures supported (scalar)

Supported in the Armv8.1 architecture and later.

### Usage

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD and FP register with the value of a vector element of the second source SIMD and FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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.

The following table shows the valid specifier combinations:

**Table D5-20 SQRDMLSH (Scalar) specifier combinations**

V	Ts	index
H	H	0 to 7
S	S	0 to 3

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.78 SQRDMLSH (scalar)

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

### Syntax

SQRDMLSH *Vd, Vn,Vm*

Where:

*v*

Is a width specifier, and can be either H or S.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.1 architecture and later.

### Usage

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD and FP register with the corresponding vector elements of the second source SIMD and FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.79 SQRDMULH (scalar, by element)

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

### Syntax

`SQRDMULH Vd, Vn, Vm.Ts[index]`

Where:

*v*

Is a width specifier, and can be either H or S.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*vm*

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

*Ts*

Is an element size specifier, and can be either H or S.

*index*

Is the element index, in the range shown in Usage.

### Usage

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see [D5.70 SQDMULH \(scalar, by element\) on page D5-1188](#).

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D5-21 SQRDMULH (Scalar) specifier combinations**

<i>v</i>	<i>Ts</i>	<i>index</i>
H	H	0 to 7
S	S	0 to 3

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.80 SQRDMULH (scalar)

Signed saturating Rounding Doubling Multiply returning High half.

### Syntax

SQRDMULH *Vd*, *Vn*, *Vm*

Where:

*v*

Is a width specifier, and can be either H or S.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Usage

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD and FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see [D5.71 SQDMULH \(scalar\) on page D5-1189](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.81 SQRSHL (scalar)

Signed saturating Rounding Shift Left (register).

### Syntax

SQRSHL  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, and can be one of B, H, S or D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD and FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see [D5.85 SQSHL \(scalar, register\) on page D5-1203](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.82 SQRSHRN (scalar)

Signed saturating Rounded Shift Right Narrow (immediate).

### Syntax

`SQRSHRN Vbd, Van, #shift`

Where:

***Vb***

Is the destination width specifier, and can be one of the values shown in Usage.

***d***

Is the number of the SIMD and FP destination register.

***Va***

Is the source width specifier, and can be one of the values shown in Usage.

***n***

Is the number of the first SIMD and FP source register.

***shift***

Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

### Usage

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see [D5.87 SQSHRN \(scalar\) on page D5-1205](#).

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

Table D5-22 SQRSHRN (Scalar) specifier combinations

<b><i>Vb</i></b>	<b><i>Va</i></b>	<b><i>shift</i></b>
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.83 SQRSHRUN (scalar)

Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

### Syntax

`SQRSHRUN Vbd, Van, #shift`

Where:

***Vb***

Is the destination width specifier, and can be one of the values shown in Usage.

***d***

Is the number of the SIMD and FP destination register.

***Va***

Is the source width specifier, and can be one of the values shown in Usage.

***n***

Is the number of the first SIMD and FP source register.

***shift***

Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

### Usage

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD and FP register. The results are rounded. For truncated results, see [D5.88 SQSHRUN \(scalar\) on page D5-1206](#).

The SQRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D5-23 SQRSHRUN (Scalar) specifier combinations**

<b><i>Vb</i></b>	<b><i>Va</i></b>	<b><i>shift</i></b>
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.84 SQSHL (scalar, immediate)

Signed saturating Shift Left (immediate).

### Syntax

SQSHL *Vd, Vn, #shift*

Where:

*v*

Is a width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the left shift amount, in the range 0 to the operand width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD and FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see [D5.104 UQRSHL \(scalar\) on page D5-1222](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

Table D5-24 SQSHL (Scalar) specifier combinations

<i>v</i>	<i>shift</i>
B	0 to 7
H	0 to 15
S	0 to 31
D	0 to 63

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.85 SQSHL (scalar, register)

Signed saturating Shift Left (register).

### Syntax

SQSHL  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, and can be one of B, H, S or D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see [D5.81 SQRSHL \(scalar\) on page D5-1199](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.86 SQSHLU (scalar)

Signed saturating Shift Left Unsigned (immediate).

### Syntax

SQSHLU *Vd, vn, #shift*

Where:

*v*

Is a width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the left shift amount, in the range 0 to the operand width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see [D5.104 UQRSHL \(scalar\) on page D5-1222](#).

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

Table D5-25 SQSHLU (Scalar) specifier combinations

<i>v</i>	<i>shift</i>
B	0 to 7
H	0 to 15
S	0 to 31
D	0 to 63

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.87 SQSHRN (scalar)

Signed saturating Shift Right Narrow (immediate).

### Syntax

`SQSHRN Vbd, Van, #shift`

Where:

***Vb***

Is the destination width specifier, and can be one of the values shown in Usage.

***d***

Is the number of the SIMD and FP destination register.

***Va***

Is the source width specifier, and can be one of the values shown in Usage.

***n***

Is the number of the first SIMD and FP source register.

***shift***

Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

### Usage

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see [D5.82 SQRSHRN \(scalar\) on page D5-1200](#).

The SQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D5-26 SQSHRN (Scalar) specifier combinations**

<b><i>Vb</i></b>	<b><i>Va</i></b>	<b><i>shift</i></b>
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.88 SQSHRUN (scalar)

Signed saturating Shift Right Unsigned Narrow (immediate).

### Syntax

`SQSHRUN Vbd, Van, #shift`

Where:

***Vb***

Is the destination width specifier, and can be one of the values shown in Usage.

***d***

Is the number of the SIMD and FP destination register.

***Va***

Is the source width specifier, and can be one of the values shown in Usage.

***n***

Is the number of the first SIMD and FP source register.

***shift***

Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

### Usage

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see [D5.83 SQRSHRUN \(scalar\) on page D5-1201](#).

The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D5-27 SQSHRUN (Scalar) specifier combinations**

<b><i>Vb</i></b>	<b><i>Va</i></b>	<b><i>shift</i></b>
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.89 SQSUB (scalar)

Signed saturating Subtract.

### Syntax

SQSUB *Vd*, *Vn*, *Vm*

Where:

*v*

Is a width specifier, and can be one of B, H, S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Usage

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD and FP register from the corresponding element values of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.90 SQXTN (scalar)

Signed saturating extract Narrow.

### Syntax

`SQXTN Vbd, Van`

Where:

**Vb**

Is the destination width specifier, and can be one of the values shown in Usage.

**d**

Is the number of the SIMD and FP destination register.

**Va**

Is the source width specifier, and can be one of the values shown in Usage.

**n**

Is the number of the SIMD and FP source register.

### Usage

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD and FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. 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 SQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTN2 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.

The following table shows the valid specifier combinations:

**Table D5-28 SQXTN (Scalar) specifier combinations**

<b>Vb</b>	<b>Va</b>
B	H
H	S
S	D

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.91 SQXTUN (scalar)

Signed saturating extract Unsigned Narrow.

### Syntax

SQXTUN *Vbd*, *Van*

Where:

*Vb*

Is the destination width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*Va*

Is the source width specifier, and can be one of the values shown in Usage.

*n*

Is the number of the SIMD and FP source register.

### Usage

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD and FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQXTUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTUN2 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.

The following table shows the valid specifier combinations:

Table D5-29 SQXTUN (Scalar) specifier combinations

<i>Vb</i>	<i>Va</i>
B	H
H	S
S	D

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.92 SRI (scalar)

Shift Right and Insert (immediate).

### Syntax

`SRI Vd, Vn, #shift`

Where:

**v**

Is a width specifier, D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**shift**

Is the right shift amount, in the range 1 to 64.

### Usage

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD and FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.93 SRSHL (scalar)

Signed Rounding Shift Left (register).

### Syntax

SRSHL  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD and FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see [D5.96 SSHL \(scalar\) on page D5-1214](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.94 SRSHR (scalar)

Signed Rounding Shift Right (immediate).

### Syntax

SRSHR *Vd, Vn, #shift*

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to 64.

### Usage

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see [D5.97 SSHR \(scalar\) on page D5-1215](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.95 SRSRA (scalar)

Signed Rounding Shift Right and Accumulate (immediate).

### Syntax

SRSRA *Vd, Vn, #shift*

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to 64.

### Usage

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see [D5.98 SSRA \(scalar\) on page D5-1216](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.96 SSHL (scalar)

Signed Shift Left (register).

### Syntax

SSHL  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD and FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see [D5.93 SRSHL \(scalar\) on page D5-1211](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.97 SSHR (scalar)

Signed Shift Right (immediate).

### Syntax

SSHR *Vd*, *Vn*, #*shift*

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to 64.

### Usage

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see [D5.94 SRSHR \(scalar\) on page D5-1212](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.98 SSRA (scalar)

Signed Shift Right and Accumulate (immediate).

### Syntax

`SSRA Vd, Vn, #shift`

Where:

**v**

Is a width specifier, D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**shift**

Is the right shift amount, in the range 1 to 64.

### Usage

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see [D5.95 SRSRA \(scalar\) on page D5-1213](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.99 SUB (scalar)

Subtract (vector).

### Syntax

SUB  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Subtract (vector). This instruction subtracts each vector element in the second source SIMD and FP register from the corresponding vector element in the first source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.100 SUQADD (scalar)

Signed saturating Accumulate of Unsigned value.

### Syntax

SUQADD *Vd, Vn*

Where:

*v*

Is a width specifier, and can be one of B, H, S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Usage

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD and FP register to corresponding signed integer values of the vector elements in the destination SIMD and FP register, and writes the resulting signed integer values to the destination SIMD and 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.

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.

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

## D5.101 UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector).

### Syntax

`UCVTF Vd, Vn, #fbits`

Where:

**v**

Is a width specifier, and can be one of the values shown in Usage.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**fbits**

Is the number of fractional bits, in the range 1 to the operand width.

### Usage

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D5-30 UCVTF (Scalar) specifier combinations**

V	fbits
H	
S	1 to 32
D	1 to 64

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

### Related information

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D5.102 UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (vector).

### Syntax

```
UCVTF Hd, Hn ; Scalar half precision  
UCVTF Vd, Vn ; Scalar single-precision and double-precision
```

Where:

- Hd** Is the 16-bit name of the SIMD and FP destination register.
- Hn** Is the 16-bit name of the SIMD and FP source register.
- V** Is a width specifier, and can be either S or D.
- d** Is the number of the SIMD and FP destination register.
- n** Is the number of the SIMD and FP source register.

### Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

### Usage

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D5.103 UQADD (scalar)

Unsigned saturating Add.

### Syntax

UQADD  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, and can be one of B, H, S or D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.104 UQRSHL (scalar)

Unsigned saturating Rounding Shift Left (register).

### Syntax

UQRSHL  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, and can be one of B, H, S or D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD and FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see [D5.106 UQSHL \(scalar, immediate\) on page D5-1224](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.105 UQRSHRN (scalar)

Unsigned saturating Rounded Shift Right Narrow (immediate).

### Syntax

`UQRSHRN Vbd, Van, #shift`

Where:

***Vb***

Is the destination width specifier, and can be one of the values shown in Usage.

***d***

Is the number of the SIMD and FP destination register.

***Va***

Is the source width specifier, and can be one of the values shown in Usage.

***n***

Is the number of the first SIMD and FP source register.

***shift***

Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

### Usage

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [D5.108 UQSHRN \(scalar\) on page D5-1226](#).

The UQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D5-31 UQRSHRN (Scalar) specifier combinations**

<b><i>Vb</i></b>	<b><i>Va</i></b>	<b><i>shift</i></b>
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.106 UQSHL (scalar, immediate)

Unsigned saturating Shift Left (immediate).

### Syntax

`UQSHL Vd, Vn, #shift`

Where:

**v**

Is a width specifier, and can be one of the values shown in Usage.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**shift**

Is the left shift amount, in the range 0 to the operand width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD and FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see

[D5.104 UQRSHL \(scalar\) on page D5-1222](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D5-32 UQSHL (Scalar) specifier combinations**

V	shift
B	0 to 7
H	0 to 15
S	0 to 31
D	0 to 63

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.107 UQSHL (scalar, register)

Unsigned saturating Shift Left (register).

### Syntax

UQSHL  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, and can be one of B, H, S or D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see [D5.104 UQRSHL \(scalar\) on page D5-1222](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.108 UQSHRN (scalar)

Unsigned saturating Shift Right Narrow (immediate).

### Syntax

`UQSHRN Vbd, Van, #shift`

Where:

**Vb**

Is the destination width specifier, and can be one of the values shown in Usage.

**d**

Is the number of the SIMD and FP destination register.

**Va**

Is the source width specifier, and can be one of the values shown in Usage.

**n**

Is the number of the first SIMD and FP source register.

**shift**

Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

### Usage

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [D5.105 UQRSHRN \(scalar\) on page D5-1223](#).

The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D5-33 UQSHRN (Scalar) specifier combinations**

<b>Vb</b>	<b>Va</b>	<b>shift</b>
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.109 UQSUB (scalar)

Unsigned saturating Subtract.

### Syntax

UQSUB *Vd*, *Vn*, *Vm*

Where:

*v*

Is a width specifier, and can be one of B, H, S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*m*

Is the number of the second SIMD and FP source register.

### Usage

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD and FP register from the corresponding element values of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

## D5.110 UQXTN (scalar)

Unsigned saturating extract Narrow.

### Syntax

UQXTN *Vbd*, *Van*

Where:

*Vb*

Is the destination width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*Va*

Is the source width specifier, and can be one of the values shown in Usage.

*n*

Is the number of the SIMD and FP source register.

### Usage

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD and FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The UQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQXTN2 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.

The following table shows the valid specifier combinations:

Table D5-34 UQXTN (Scalar) specifier combinations

<i>Vb</i>	<i>Va</i>
B	H
H	S
S	D

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.111 URSHL (scalar)

Unsigned Rounding Shift Left (register).

### Syntax

URSHL  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

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

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

## D5.112 URSHR (scalar)

Unsigned Rounding Shift Right (immediate).

### Syntax

URSHR *Vd, Vn, #shift*

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to 64.

### Usage

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [D5.115 USHR \(scalar\) on page D5-1233](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.113 URSRA (scalar)

Unsigned Rounding Shift Right and Accumulate (immediate).

### Syntax

URSRA *Vd, Vn, #shift*

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to 64.

### Usage

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [D5.117 USRA \(scalar\) on page D5-1235](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D5.114 USHL (scalar)

Unsigned Shift Left (register).

### Syntax

USHL  $Vd$ ,  $Vn$ ,  $Vm$

Where:

$v$

Is a width specifier, D.

$d$

Is the number of the SIMD and FP destination register.

$n$

Is the number of the first SIMD and FP source register.

$m$

Is the number of the second SIMD and FP source register.

### Usage

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see [D5.111 URSHL \(scalar\) on page D5-1229](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.115 USHR (scalar)

Unsigned Shift Right (immediate).

### Syntax

USHR *Vd*, *Vn*, #*shift*

Where:

*v*

Is a width specifier, D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the first SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to 64.

### Usage

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [D5.112 URSHR \(scalar\) on page D5-1230](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order on page D5-1110](#)

## D5.116 USQADD (scalar)

Unsigned saturating Accumulate of Signed value.

### Syntax

USQADD *Vd, Vn*

Where:

*v*

Is a width specifier, and can be one of B, H, S or D.

*d*

Is the number of the SIMD and FP destination register.

*n*

Is the number of the SIMD and FP source register.

### Usage

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD and FP register to corresponding unsigned integer values of the vector elements in the destination SIMD and FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD and 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.

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.

### Related reference

*D5.1 A64 SIMD scalar instructions in alphabetical order* on page D5-1110

## D5.117 USRA (scalar)

Unsigned Shift Right and Accumulate (immediate).

### Syntax

**USRA *Vd*, *Vn*, #*shift***

Where:

**v**

Is a width specifier, D.

**d**

Is the number of the SIMD and FP destination register.

**n**

Is the number of the first SIMD and FP source register.

**shift**

Is the right shift amount, in the range 1 to 64.

### Usage

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [D5.113 URSRA \(scalar\) on page D5-1231](#).

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.

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110



# Chapter D6

## A64 SIMD Vector Instructions

Describes the A64 SIMD vector instructions.

It contains the following sections:

- [\*D6.1 A64 SIMD Vector instructions in alphabetical order\*](#) on page D6-1243.
- [\*D6.2 ABS \(vector\)\*](#) on page D6-1254.
- [\*D6.3 ADD \(vector\)\*](#) on page D6-1255.
- [\*D6.4 ADDHN, ADDHN2 \(vector\)\*](#) on page D6-1256.
- [\*D6.5 ADDP \(vector\)\*](#) on page D6-1257.
- [\*D6.6 ADDV \(vector\)\*](#) on page D6-1258.
- [\*D6.7 AND \(vector\)\*](#) on page D6-1259.
- [\*D6.8 BIC \(vector, immediate\)\*](#) on page D6-1260.
- [\*D6.9 BIC \(vector, register\)\*](#) on page D6-1261.
- [\*D6.10 BIF \(vector\)\*](#) on page D6-1262.
- [\*D6.11 BIT \(vector\)\*](#) on page D6-1263.
- [\*D6.12 BSL \(vector\)\*](#) on page D6-1264.
- [\*D6.13 CLS \(vector\)\*](#) on page D6-1265.
- [\*D6.14 CLZ \(vector\)\*](#) on page D6-1266.
- [\*D6.15 CMEQ \(vector, register\)\*](#) on page D6-1267.
- [\*D6.16 CMEQ \(vector, zero\)\*](#) on page D6-1268.
- [\*D6.17 CMGE \(vector, register\)\*](#) on page D6-1269.
- [\*D6.18 CMGE \(vector, zero\)\*](#) on page D6-1270.
- [\*D6.19 CMGT \(vector, register\)\*](#) on page D6-1271.
- [\*D6.20 CMGT \(vector, zero\)\*](#) on page D6-1272.
- [\*D6.21 CMHI \(vector, register\)\*](#) on page D6-1273.
- [\*D6.22 CMHS \(vector, register\)\*](#) on page D6-1274.
- [\*D6.23 CMLE \(vector, zero\)\*](#) on page D6-1275.

- *D6.24 CMLT (vector, zero)* on page D6-1276.
- *D6.25 CMTST (vector)* on page D6-1277.
- *D6.26 CNT (vector)* on page D6-1278.
- *D6.27 DUP (vector, element)* on page D6-1279.
- *D6.28 DUP (vector, general)* on page D6-1280.
- *D6.29 EOR (vector)* on page D6-1281.
- *D6.30 EXT (vector)* on page D6-1282.
- *D6.31 FABD (vector)* on page D6-1283.
- *D6.32 FABS (vector)* on page D6-1284.
- *D6.33 FACGE (vector)* on page D6-1285.
- *D6.34 FACGT (vector)* on page D6-1286.
- *D6.35 FADD (vector)* on page D6-1287.
- *D6.36 FADDP (vector)* on page D6-1288.
- *D6.37 FCADD (vector)* on page D6-1289.
- *D6.38 FCMEQ (vector, register)* on page D6-1290.
- *D6.39 FCMEQ (vector, zero)* on page D6-1291.
- *D6.40 FCMGE (vector, register)* on page D6-1292.
- *D6.41 FCMGE (vector, zero)* on page D6-1293.
- *D6.42 FCMGT (vector, register)* on page D6-1294.
- *D6.43 FCMGT (vector, zero)* on page D6-1295.
- *D6.44 FCMLA (vector)* on page D6-1296.
- *D6.45 FCMLE (vector, zero)* on page D6-1297.
- *D6.46 FCMLT (vector, zero)* on page D6-1298.
- *D6.47 FCVTAS (vector)* on page D6-1299.
- *D6.48 FCVTAU (vector)* on page D6-1300.
- *D6.49 FCVTL, FCVTL2 (vector)* on page D6-1301.
- *D6.50 FCVTMS (vector)* on page D6-1302.
- *D6.51 FCVTMU (vector)* on page D6-1303.
- *D6.52 FCVTN, FCVTN2 (vector)* on page D6-1304.
- *D6.53 FCVTNS (vector)* on page D6-1305.
- *D6.54 FCVTNU (vector)* on page D6-1306.
- *D6.55 FCVTPS (vector)* on page D6-1307.
- *D6.56 FCVTPU (vector)* on page D6-1308.
- *D6.57 FCVTXN, FCVTXN2 (vector)* on page D6-1309.
- *D6.58 FCVTZS (vector, fixed-point)* on page D6-1310.
- *D6.59 FCVTZS (vector, integer)* on page D6-1311.
- *D6.60 FCVTZU (vector, fixed-point)* on page D6-1312.
- *D6.61 FCVTZU (vector, integer)* on page D6-1313.
- *D6.62 FDIV (vector)* on page D6-1314.
- *D6.63 FMAX (vector)* on page D6-1315.
- *D6.64 FMAXNM (vector)* on page D6-1316.
- *D6.65 FMAXNMP (vector)* on page D6-1317.
- *D6.66 FMAXNMV (vector)* on page D6-1318.
- *D6.67 FMAXP (vector)* on page D6-1319.
- *D6.68 FMAXV (vector)* on page D6-1320.
- *D6.69 FMIN (vector)* on page D6-1321.
- *D6.70 FMINNM (vector)* on page D6-1322.
- *D6.71 FMINNMP (vector)* on page D6-1323.
- *D6.72 FMINNMV (vector)* on page D6-1324.
- *D6.73 FMINP (vector)* on page D6-1325.
- *D6.74 FMINV (vector)* on page D6-1326.
- *D6.75 FMLA (vector, by element)* on page D6-1327.
- *D6.76 FMLA (vector)* on page D6-1329.
- *D6.77 FMLAL, (vector)* on page D6-1330.
- *D6.78 FMLS (vector, by element)* on page D6-1331.
- *D6.79 FMLS (vector)* on page D6-1333.

- *D6.80 FMLSL, (vector)* on page D6-1334.
- *D6.81 FMOV (vector, immediate)* on page D6-1335.
- *D6.82 FMUL (vector, by element)* on page D6-1337.
- *D6.83 FMUL (vector)* on page D6-1339.
- *D6.84 FMULX (vector, by element)* on page D6-1340.
- *D6.85 FMULX (vector)* on page D6-1342.
- *D6.86 FNEG (vector)* on page D6-1343.
- *D6.87 FRECPE (vector)* on page D6-1344.
- *D6.88 FRECPSEN (vector)* on page D6-1345.
- *D6.89 FRECPX (vector)* on page D6-1346.
- *D6.90 FRINTA (vector)* on page D6-1347.
- *D6.91 FRINTI (vector)* on page D6-1348.
- *D6.92 FRINTM (vector)* on page D6-1349.
- *D6.93 FRINTN (vector)* on page D6-1350.
- *D6.94 FRINTP (vector)* on page D6-1351.
- *D6.95 FRINTX (vector)* on page D6-1352.
- *D6.96 FRINTZ (vector)* on page D6-1353.
- *D6.97 FRSQRTE (vector)* on page D6-1354.
- *D6.98 FRSQRTS (vector)* on page D6-1355.
- *D6.99 FSQRT (vector)* on page D6-1356.
- *D6.100 FSUB (vector)* on page D6-1357.
- *D6.101 INS (vector, element)* on page D6-1358.
- *D6.102 INS (vector, general)* on page D6-1359.
- *D6.103 LD1 (vector, multiple structures)* on page D6-1360.
- *D6.104 LD1 (vector, single structure)* on page D6-1363.
- *D6.105 LD1R (vector)* on page D6-1364.
- *D6.106 LD2 (vector, multiple structures)* on page D6-1365.
- *D6.107 LD2 (vector, single structure)* on page D6-1366.
- *D6.108 LD2R (vector)* on page D6-1367.
- *D6.109 LD3 (vector, multiple structures)* on page D6-1368.
- *D6.110 LD3 (vector, single structure)* on page D6-1369.
- *D6.111 LD3R (vector)* on page D6-1371.
- *D6.112 LD4 (vector, multiple structures)* on page D6-1372.
- *D6.113 LD4 (vector, single structure)* on page D6-1373.
- *D6.114 LD4R (vector)* on page D6-1375.
- *D6.115 MLA (vector, by element)* on page D6-1376.
- *D6.116 MLA (vector)* on page D6-1377.
- *D6.117 MLS (vector, by element)* on page D6-1378.
- *D6.118 MLS (vector)* on page D6-1379.
- *D6.119 MOV (vector, element)* on page D6-1380.
- *D6.120 MOV (vector, from general)* on page D6-1381.
- *D6.121 MOV (vector)* on page D6-1382.
- *D6.122 MOV (vector, to general)* on page D6-1383.
- *D6.123 MOVI (vector)* on page D6-1384.
- *D6.124 MUL (vector, by element)* on page D6-1386.
- *D6.125 MUL (vector)* on page D6-1387.
- *D6.126 MVN (vector)* on page D6-1388.
- *D6.127 MVNI (vector)* on page D6-1389.
- *D6.128 NEG (vector)* on page D6-1390.
- *D6.129 NOT (vector)* on page D6-1391.
- *D6.130 ORN (vector)* on page D6-1392.
- *D6.131 ORR (vector, immediate)* on page D6-1393.
- *D6.132 ORR (vector, register)* on page D6-1394.
- *D6.133 PMUL (vector)* on page D6-1395.
- *D6.134 PMULL, PMULL2 (vector)* on page D6-1396.
- *D6.135 RADDHN, RADDHN2 (vector)* on page D6-1397.

- [D6.136 RBIT \(vector\)](#) on page D6-1398.
- [D6.137 REV16 \(vector\)](#) on page D6-1399.
- [D6.138 REV32 \(vector\)](#) on page D6-1400.
- [D6.139 REV64 \(vector\)](#) on page D6-1401.
- [D6.140 RSHRN, RSHRN2 \(vector\)](#) on page D6-1402.
- [D6.141 RSUBHN, RSUBHN2 \(vector\)](#) on page D6-1403.
- [D6.142 SABA \(vector\)](#) on page D6-1404.
- [D6.143 SABAL, SABAL2 \(vector\)](#) on page D6-1405.
- [D6.144 SABD \(vector\)](#) on page D6-1406.
- [D6.145 SABDL, SABDL2 \(vector\)](#) on page D6-1407.
- [D6.146 SADALP \(vector\)](#) on page D6-1408.
- [D6.147 SADDL, SADDL2 \(vector\)](#) on page D6-1409.
- [D6.148 SADDLP \(vector\)](#) on page D6-1410.
- [D6.149 SADDLV \(vector\)](#) on page D6-1411.
- [D6.150 SADDW, SADDW2 \(vector\)](#) on page D6-1412.
- [D6.151 SCVT<sub>F</sub> \(vector, fixed-point\)](#) on page D6-1413.
- [D6.152 SCVT<sub>F</sub> \(vector, integer\)](#) on page D6-1414.
- [D6.153 SDOT \(vector, by element\)](#) on page D6-1415.
- [D6.154 SDOT \(vector\)](#) on page D6-1416.
- [D6.155 SHADD \(vector\)](#) on page D6-1417.
- [D6.156 SHL \(vector\)](#) on page D6-1418.
- [D6.157 SHLL, SHLL2 \(vector\)](#) on page D6-1419.
- [D6.158 SHRN, SHRN2 \(vector\)](#) on page D6-1420.
- [D6.159 SHSUB \(vector\)](#) on page D6-1421.
- [D6.160 SLI \(vector\)](#) on page D6-1422.
- [D6.161 SMAX \(vector\)](#) on page D6-1423.
- [D6.162 SMAXP \(vector\)](#) on page D6-1424.
- [D6.163 SMAXV \(vector\)](#) on page D6-1425.
- [D6.164 SMIN \(vector\)](#) on page D6-1426.
- [D6.165 SMINP \(vector\)](#) on page D6-1427.
- [D6.166 SMINV \(vector\)](#) on page D6-1428.
- [D6.167 SMLAL, SMLAL2 \(vector, by element\)](#) on page D6-1429.
- [D6.168 SMLAL, SMLAL2 \(vector\)](#) on page D6-1430.
- [D6.169 SMLS<sub>L</sub>, SMLS<sub>L</sub>2 \(vector, by element\)](#) on page D6-1431.
- [D6.170 SMLS<sub>L</sub>, SMLS<sub>L</sub>2 \(vector\)](#) on page D6-1432.
- [D6.171 SMOV \(vector\)](#) on page D6-1433.
- [D6.172 SMULL, SMULL2 \(vector, by element\)](#) on page D6-1434.
- [D6.173 SMULL, SMULL2 \(vector\)](#) on page D6-1435.
- [D6.174 SQABS \(vector\)](#) on page D6-1436.
- [D6.175 SQADD \(vector\)](#) on page D6-1437.
- [D6.176 SQDMLAL, SQDMLAL2 \(vector, by element\)](#) on page D6-1438.
- [D6.177 SQDMLAL, SQDMLAL2 \(vector\)](#) on page D6-1440.
- [D6.178 SQDMLSL, SQDMLSL2 \(vector, by element\)](#) on page D6-1441.
- [D6.179 SQDMLSL, SQDMLSL2 \(vector\)](#) on page D6-1443.
- [D6.180 SQDMULH \(vector, by element\)](#) on page D6-1444.
- [D6.181 SQDMULH \(vector\)](#) on page D6-1445.
- [D6.182 SQDMULL, SQDMULL2 \(vector, by element\)](#) on page D6-1446.
- [D6.183 SQDMULL, SQDMULL2 \(vector\)](#) on page D6-1448.
- [D6.184 SQNEG \(vector\)](#) on page D6-1449.
- [D6.185 SQRDMLAH \(vector, by element\)](#) on page D6-1450.
- [D6.186 SQRDMLAH \(vector\)](#) on page D6-1451.
- [D6.187 SQRDMLSH \(vector, by element\)](#) on page D6-1452.
- [D6.188 SQRDMLSH \(vector\)](#) on page D6-1453.
- [D6.189 SQRDMULH \(vector, by element\)](#) on page D6-1454.
- [D6.190 SQRDMULH \(vector\)](#) on page D6-1455.
- [D6.191 SQRSHL \(vector\)](#) on page D6-1456.

- [D6.192 SQRSHRN, SQRSHRN2 \(vector\)](#) on page D6-1457.
- [D6.193 SQRSHRUN, SQRSHRUN2 \(vector\)](#) on page D6-1458.
- [D6.194 SQSHL \(vector, immediate\)](#) on page D6-1459.
- [D6.195 SQSHL \(vector, register\)](#) on page D6-1460.
- [D6.196 SQSHLU \(vector\)](#) on page D6-1461.
- [D6.197 SQSHRN, SQSHRN2 \(vector\)](#) on page D6-1462.
- [D6.198 SQSHRUN, SQSHRUN2 \(vector\)](#) on page D6-1463.
- [D6.199 SQSUB \(vector\)](#) on page D6-1464.
- [D6.200 SQXTN, SQXTN2 \(vector\)](#) on page D6-1465.
- [D6.201 SQXTUN, SQXTUN2 \(vector\)](#) on page D6-1466.
- [D6.202 SRHADD \(vector\)](#) on page D6-1467.
- [D6.203 SRI \(vector\)](#) on page D6-1468.
- [D6.204 SRSHL \(vector\)](#) on page D6-1469.
- [D6.205 SRSHR \(vector\)](#) on page D6-1470.
- [D6.206 SRSRA \(vector\)](#) on page D6-1471.
- [D6.207 SSHL \(vector\)](#) on page D6-1472.
- [D6.208 SSHLL, SSHLL2 \(vector\)](#) on page D6-1473.
- [D6.209 SSHR \(vector\)](#) on page D6-1474.
- [D6.210 SSRA \(vector\)](#) on page D6-1475.
- [D6.211 SSUBL, SSUBL2 \(vector\)](#) on page D6-1476.
- [D6.212 SSUBW, SSUBW2 \(vector\)](#) on page D6-1477.
- [D6.213 ST1 \(vector, multiple structures\)](#) on page D6-1478.
- [D6.214 ST1 \(vector, single structure\)](#) on page D6-1481.
- [D6.215 ST2 \(vector, multiple structures\)](#) on page D6-1482.
- [D6.216 ST2 \(vector, single structure\)](#) on page D6-1483.
- [D6.217 ST3 \(vector, multiple structures\)](#) on page D6-1484.
- [D6.218 ST3 \(vector, single structure\)](#) on page D6-1485.
- [D6.219 ST4 \(vector, multiple structures\)](#) on page D6-1487.
- [D6.220 ST4 \(vector, single structure\)](#) on page D6-1488.
- [D6.221 SUB \(vector\)](#) on page D6-1490.
- [D6.222 SUBHN, SUBHN2 \(vector\)](#) on page D6-1491.
- [D6.223 SUQADD \(vector\)](#) on page D6-1492.
- [D6.224 SXTL, SXTL2 \(vector\)](#) on page D6-1493.
- [D6.225 TBL \(vector\)](#) on page D6-1494.
- [D6.226 TBX \(vector\)](#) on page D6-1495.
- [D6.227 TRN1 \(vector\)](#) on page D6-1496.
- [D6.228 TRN2 \(vector\)](#) on page D6-1497.
- [D6.229 UABA \(vector\)](#) on page D6-1498.
- [D6.230 UABAL, UABAL2 \(vector\)](#) on page D6-1499.
- [D6.231 UABD \(vector\)](#) on page D6-1500.
- [D6.232 UABDL, UABDL2 \(vector\)](#) on page D6-1501.
- [D6.233 UADALP \(vector\)](#) on page D6-1502.
- [D6.234 UADDL, UADDL2 \(vector\)](#) on page D6-1503.
- [D6.235 UADDLP \(vector\)](#) on page D6-1504.
- [D6.236 UADDLV \(vector\)](#) on page D6-1505.
- [D6.237 UADDW, UADDW2 \(vector\)](#) on page D6-1506.
- [D6.238 UCVTF \(vector, fixed-point\)](#) on page D6-1507.
- [D6.239 UCVTF \(vector, integer\)](#) on page D6-1508.
- [D6.240 UDOT \(vector, by element\)](#) on page D6-1509.
- [D6.241 UDOT \(vector\)](#) on page D6-1510.
- [D6.242 UHADD \(vector\)](#) on page D6-1511.
- [D6.243 UHSUB \(vector\)](#) on page D6-1512.
- [D6.244 UMAX \(vector\)](#) on page D6-1513.
- [D6.245 UMAXP \(vector\)](#) on page D6-1514.
- [D6.246 UMAXV \(vector\)](#) on page D6-1515.
- [D6.247 UMIN \(vector\)](#) on page D6-1516.

- *D6.248 UMINP (vector)* on page D6-1517.
- *D6.249 UMINV (vector)* on page D6-1518.
- *D6.250 UMLAL, UMLAL2 (vector, by element)* on page D6-1519.
- *D6.251 UMLAL, UMLAL2 (vector)* on page D6-1520.
- *D6.252 UMLSL, UMLSL2 (vector, by element)* on page D6-1521.
- *D6.253 UMLSL, UMLSL2 (vector)* on page D6-1522.
- *D6.254 UMOV (vector)* on page D6-1523.
- *D6.255 UMULL, UMULL2 (vector, by element)* on page D6-1524.
- *D6.256 UMULL, UMULL2 (vector)* on page D6-1525.
- *D6.257 UQADD (vector)* on page D6-1526.
- *D6.258 UQRSHL (vector)* on page D6-1527.
- *D6.259 UQRSHRN, UQRSHRN2 (vector)* on page D6-1528.
- *D6.260 UQSHL (vector, immediate)* on page D6-1529.
- *D6.261 UQSHL (vector, register)* on page D6-1530.
- *D6.262 UQSHRN, UQSHRN2 (vector)* on page D6-1531.
- *D6.263 UQSUB (vector)* on page D6-1533.
- *D6.264 UQXTN, UQXTN2 (vector)* on page D6-1534.
- *D6.265 URECPE (vector)* on page D6-1535.
- *D6.266 URHADD (vector)* on page D6-1536.
- *D6.267 URSHL (vector)* on page D6-1537.
- *D6.268 URSHR (vector)* on page D6-1538.
- *D6.269 URSQRT (vector)* on page D6-1539.
- *D6.270 URSRA (vector)* on page D6-1540.
- *D6.271 USHL (vector)* on page D6-1541.
- *D6.272 USHLL, USHLL2 (vector)* on page D6-1542.
- *D6.273 USHR (vector)* on page D6-1543.
- *D6.274 USQADD (vector)* on page D6-1544.
- *D6.275 USRA (vector)* on page D6-1545.
- *D6.276 USUBL, USUBL2 (vector)* on page D6-1546.
- *D6.277 USUBW, USUBW2 (vector)* on page D6-1547.
- *D6.278 UXTL, UXTL2 (vector)* on page D6-1548.
- *D6.279 UZP1 (vector)* on page D6-1549.
- *D6.280 UZP2 (vector)* on page D6-1550.
- *D6.281 XTN, XTN2 (vector)* on page D6-1551.
- *D6.282 ZIP1 (vector)* on page D6-1552.
- *D6.283 ZIP2 (vector)* on page D6-1553.

## D6.1 A64 SIMD Vector instructions in alphabetical order

A summary of the A64 SIMD Vector instructions that are supported.

**Table D6-1 Summary of A64 SIMD Vector instructions**

Mnemonic	Brief description	See
ABS (vector)	Absolute value (vector)	<a href="#">D6.2 ABS (vector) on page D6-1254</a>
ADD (vector)	Add (vector)	<a href="#">D6.3 ADD (vector) on page D6-1255</a>
ADDHN, ADDHN2 (vector)	Add returning High Narrow	<a href="#">D6.4 ADDHN, ADDHN2 (vector) on page D6-1256</a>
ADDP (vector)	Add Pairwise (vector)	<a href="#">D6.5 ADDP (vector) on page D6-1257</a>
ADDV (vector)	Add across Vector	<a href="#">D6.6 ADDV (vector) on page D6-1258</a>
AND (vector)	Bitwise AND (vector)	<a href="#">D6.7 AND (vector) on page D6-1259</a>
BIC (vector, immediate)	Bitwise bit Clear (vector, immediate)	<a href="#">D6.8 BIC (vector; immediate) on page D6-1260</a>
BIC (vector, register)	Bitwise bit Clear (vector, register)	<a href="#">D6.9 BIC (vector; register) on page D6-1261</a>
BIF (vector)	Bitwise Insert if False	<a href="#">D6.10 BIF (vector) on page D6-1262</a>
BIT (vector)	Bitwise Insert if True	<a href="#">D6.11 BIT (vector) on page D6-1263</a>
BSL (vector)	Bitwise Select	<a href="#">D6.12 BSL (vector) on page D6-1264</a>
CLS (vector)	Count Leading Sign bits (vector)	<a href="#">D6.13 CLS (vector) on page D6-1265</a>
CLZ (vector)	Count Leading Zero bits (vector)	<a href="#">D6.14 CLZ (vector) on page D6-1266</a>
CMEQ (vector, register)	Compare bitwise Equal (vector)	<a href="#">D6.15 CMEQ (vector; register) on page D6-1267</a>
CMEQ (vector, zero)	Compare bitwise Equal to zero (vector)	<a href="#">D6.16 CMEQ (vector; zero) on page D6-1268</a>
CMGE (vector, register)	Compare signed Greater than or Equal (vector)	<a href="#">D6.17 CMGE (vector; register) on page D6-1269</a>
CMGE (vector, zero)	Compare signed Greater than or Equal to zero (vector)	<a href="#">D6.18 CMGE (vector; zero) on page D6-1270</a>
CMGT (vector, register)	Compare signed Greater than (vector)	<a href="#">D6.19 CMGT (vector; register) on page D6-1271</a>
CMGT (vector, zero)	Compare signed Greater than zero (vector)	<a href="#">D6.20 CMGT (vector; zero) on page D6-1272</a>
CMHI (vector, register)	Compare unsigned Higher (vector)	<a href="#">D6.21 CMHI (vector; register) on page D6-1273</a>
CMHS (vector, register)	Compare unsigned Higher or Same (vector)	<a href="#">D6.22 CMHS (vector; register) on page D6-1274</a>
CMLE (vector, zero)	Compare signed Less than or Equal to zero (vector)	<a href="#">D6.23 CMLE (vector; zero) on page D6-1275</a>
CMLT (vector, zero)	Compare signed Less than zero (vector)	<a href="#">D6.24 CMLT (vector; zero) on page D6-1276</a>
CMTST (vector)	Compare bitwise Test bits nonzero (vector)	<a href="#">D6.25 CMTST (vector) on page D6-1277</a>
CNT (vector)	Population Count per byte	<a href="#">D6.26 CNT (vector) on page D6-1278</a>
DUP (vector, element)	vector	<a href="#">D6.27 DUP (vector; element) on page D6-1279</a>
DUP (vector, general)	Duplicate general-purpose register to vector	<a href="#">D6.28 DUP (vector; general) on page D6-1280</a>
EOR (vector)	Bitwise Exclusive OR (vector)	<a href="#">D6.29 EOR (vector) on page D6-1281</a>
EXT (vector)	Extract vector from pair of vectors	<a href="#">D6.30 EXT (vector) on page D6-1282</a>
FABD (vector)	Floating-point Absolute Difference (vector)	<a href="#">D6.31 FABD (vector) on page D6-1283</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

Mnemonic	Brief description	See
FABS (vector)	Floating-point Absolute value (vector)	<a href="#">D6.32 FABS (vector) on page D6-1284</a>
FACGE (vector)	Floating-point Absolute Compare Greater than or Equal (vector)	<a href="#">D6.33 FACGE (vector) on page D6-1285</a>
FACGT (vector)	Floating-point Absolute Compare Greater than (vector)	<a href="#">D6.34 FACGT (vector) on page D6-1286</a>
FADD (vector)	Floating-point Add (vector)	<a href="#">D6.35 FADD (vector) on page D6-1287</a>
FADDP (vector)	Floating-point Add Pairwise (vector)	<a href="#">D6.36 FADDP (vector) on page D6-1288</a>
FCADD (vector)	Floating-point Complex Add	<a href="#">D6.37 FCADD (vector) on page D6-1289</a>
FCMEQ (vector, register)	Floating-point Compare Equal (vector)	<a href="#">D6.38 FCMEQ (vector, register) on page D6-1290</a>
FCMEQ (vector, zero)	Floating-point Compare Equal to zero (vector)	<a href="#">D6.39 FCMEQ (vector, zero) on page D6-1291</a>
FCMGE (vector, register)	Floating-point Compare Greater than or Equal (vector)	<a href="#">D6.40 FCMGE (vector, register) on page D6-1292</a>
FCMGE (vector, zero)	Floating-point Compare Greater than or Equal to zero (vector)	<a href="#">D6.41 FCMGE (vector, zero) on page D6-1293</a>
FCMGT (vector, register)	Floating-point Compare Greater than (vector)	<a href="#">D6.42 FCMGT (vector, register) on page D6-1294</a>
FCMGT (vector, zero)	Floating-point Compare Greater than zero (vector)	<a href="#">D6.43 FCMGT (vector, zero) on page D6-1295</a>
FCMLA (vector)	Floating-point Complex Multiply Accumulate	<a href="#">D6.44 FCMLA (vector) on page D6-1296</a>
FCMLE (vector, zero)	Floating-point Compare Less than or Equal to zero (vector)	<a href="#">D6.45 FCMLE (vector, zero) on page D6-1297</a>
FCMLT (vector, zero)	Floating-point Compare Less than zero (vector)	<a href="#">D6.46 FCMLT (vector, zero) on page D6-1298</a>
FCVTAS (vector)	Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector)	<a href="#">D6.47 FCVTAS (vector) on page D6-1299</a>
FCVTAU (vector)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector)	<a href="#">D6.48 FCVTAU (vector) on page D6-1300</a>
FCVTL, FCVTL2 (vector)	Floating-point Convert to higher precision Long (vector)	<a href="#">D6.49 FCVTL, FCVTL2 (vector) on page D6-1301</a>
FCVTMS (vector)	Floating-point Convert to Signed integer, rounding toward Minus infinity (vector)	<a href="#">D6.50 FCVTMS (vector) on page D6-1302</a>
FCVTMU (vector)	Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector)	<a href="#">D6.51 FCVTMU (vector) on page D6-1303</a>
FCVTN, FCVTN2 (vector)	Floating-point Convert to lower precision Narrow (vector)	<a href="#">D6.52 FCVTN, FCVTN2 (vector) on page D6-1304</a>
FCVTNS (vector)	Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector)	<a href="#">D6.53 FCVTNS (vector) on page D6-1305</a>
FCVTNU (vector)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector)	<a href="#">D6.54 FCVTNU (vector) on page D6-1306</a>
FCVTPS (vector)	Floating-point Convert to Signed integer, rounding toward Plus infinity (vector)	<a href="#">D6.55 FCVTPS (vector) on page D6-1307</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
FCVTPU (vector)	Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector)	<a href="#">D6.56 FCVTPU (vector) on page D6-1308</a>
FCVTXN, FCVTXN2 (vector)	Floating-point Convert to lower precision Narrow, rounding to odd (vector)	<a href="#">D6.57 FCVTXN, FCVTXN2 (vector) on page D6-1309</a>
FCVTZS (vector, fixed-point)	Floating-point Convert to Signed fixed-point, rounding toward Zero (vector)	<a href="#">D6.58 FCVTZS (vector, fixed-point) on page D6-1310</a>
FCVTZS (vector, integer)	Floating-point Convert to Signed integer, rounding toward Zero (vector)	<a href="#">D6.59 FCVTZS (vector, integer) on page D6-1311</a>
FCVTZU (vector, fixed-point)	Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector)	<a href="#">D6.60 FCVTZU (vector, fixed-point) on page D6-1312</a>
FCVTZU (vector, integer)	Floating-point Convert to Unsigned integer, rounding toward Zero (vector)	<a href="#">D6.61 FCVTZU (vector, integer) on page D6-1313</a>
FDIV (vector)	Floating-point Divide (vector)	<a href="#">D6.62 FDIV (vector) on page D6-1314</a>
FMAX (vector)	Floating-point Maximum (vector)	<a href="#">D6.63 FMAX (vector) on page D6-1315</a>
FMAXNM (vector)	Floating-point Maximum Number (vector)	<a href="#">D6.64 FMAXNM (vector) on page D6-1316</a>
FMAXNMP (vector)	Floating-point Maximum Number Pairwise (vector)	<a href="#">D6.65 FMAXNMP (vector) on page D6-1317</a>
FMAXNMV (vector)	Floating-point Maximum Number across Vector	<a href="#">D6.66 FMAXNMV (vector) on page D6-1318</a>
FMAXP (vector)	Floating-point Maximum Pairwise (vector)	<a href="#">D6.67 FMAXP (vector) on page D6-1319</a>
FMAXV (vector)	Floating-point Maximum across Vector	<a href="#">D6.68 FMAXV (vector) on page D6-1320</a>
FMIN (vector)	Floating-point minimum (vector)	<a href="#">D6.69 FMIN (vector) on page D6-1321</a>
FMINNM (vector)	Floating-point Minimum Number (vector)	<a href="#">D6.70 FMINNM (vector) on page D6-1322</a>
FMINNMP (vector)	Floating-point Minimum Number Pairwise (vector)	<a href="#">D6.71 FMINNMP (vector) on page D6-1323</a>
FMINNMV (vector)	Floating-point Minimum Number across Vector	<a href="#">D6.72 FMINNMV (vector) on page D6-1324</a>
FMINP (vector)	Floating-point Minimum Pairwise (vector)	<a href="#">D6.73 FMINP (vector) on page D6-1325</a>
FMINV (vector)	Floating-point Minimum across Vector	<a href="#">D6.74 FMINV (vector) on page D6-1326</a>
FMLA (vector, by element)	Floating-point fused Multiply-Add to accumulator (by element)	<a href="#">D6.75 FMLA (vector, by element) on page D6-1327</a>
FMLA (vector)	Floating-point fused Multiply-Add to accumulator (vector)	<a href="#">D6.76 FMLA (vector) on page D6-1329</a>
FMLAL, (vector)	Floating-point fused Multiply-Add Long to accumulator (vector)	<a href="#">D6.77 FMLAL, (vector) on page D6-1330</a>
FMLS (vector, by element)	Floating-point fused Multiply-Subtract from accumulator (by element)	<a href="#">D6.78 FMLS (vector, by element) on page D6-1331</a>
FMLS (vector)	Floating-point fused Multiply-Subtract from accumulator (vector)	<a href="#">D6.79 FMLS (vector) on page D6-1333</a>
FMLSL, (vector)	Floating-point fused Multiply-Subtract Long from accumulator (vector)	<a href="#">D6.80 FMLSL, (vector) on page D6-1334</a>

Table D6-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
FMOV (vector, immediate)	Floating-point move immediate (vector)	<a href="#">D6.81 FMOV (vector, immediate) on page D6-1335</a>
FMUL (vector, by element)	Floating-point Multiply (by element)	<a href="#">D6.82 FMUL (vector, by element) on page D6-1337</a>
FMUL (vector)	Floating-point Multiply (vector)	<a href="#">D6.83 FMUL (vector) on page D6-1339</a>
FMULX (vector, by element)	Floating-point Multiply extended (by element)	<a href="#">D6.84 FMULX (vector, by element) on page D6-1340</a>
FMULX (vector)	Floating-point Multiply extended	<a href="#">D6.85 FMULX (vector) on page D6-1342</a>
FNEG (vector)	Floating-point Negate (vector)	<a href="#">D6.86 FNEG (vector) on page D6-1343</a>
FRECPE (vector)	Floating-point Reciprocal Estimate	<a href="#">D6.87 FRECPE (vector) on page D6-1344</a>
FRECP(S) (vector)	Floating-point Reciprocal Step	<a href="#">D6.88 FRECPS (vector) on page D6-1345</a>
FRECPX (vector)	Floating-point Reciprocal exponent (scalar)	<a href="#">D6.89 FRECPX (vector) on page D6-1346</a>
FRINTA (vector)	Floating-point Round to Integral, to nearest with ties to Away (vector)	<a href="#">D6.90 FRINTA (vector) on page D6-1347</a>
FRINTI (vector)	Floating-point Round to Integral, using current rounding mode (vector)	<a href="#">D6.91 FRINTI (vector) on page D6-1348</a>
FRINTM (vector)	Floating-point Round to Integral, toward Minus infinity (vector)	<a href="#">D6.92 FRINTM (vector) on page D6-1349</a>
FRINTN (vector)	Floating-point Round to Integral, to nearest with ties to even (vector)	<a href="#">D6.93 FRINTN (vector) on page D6-1350</a>
FRINTP (vector)	Floating-point Round to Integral, toward Plus infinity (vector)	<a href="#">D6.94 FRINTP (vector) on page D6-1351</a>
FRINTX (vector)	Floating-point Round to Integral exact, using current rounding mode (vector)	<a href="#">D6.95 FRINTX (vector) on page D6-1352</a>
FRINTZ (vector)	Floating-point Round to Integral, toward Zero (vector)	<a href="#">D6.96 FRINTZ (vector) on page D6-1353</a>
FRSQRTE (vector)	Floating-point Reciprocal Square Root Estimate	<a href="#">D6.97 FRSQRTE (vector) on page D6-1354</a>
FRSQRTS (vector)	Floating-point Reciprocal Square Root Step	<a href="#">D6.98 FRSQRTS (vector) on page D6-1355</a>
FSQRT (vector)	Floating-point Square Root (vector)	<a href="#">D6.99 FSQRT (vector) on page D6-1356</a>
FSUB (vector)	Floating-point Subtract (vector)	<a href="#">D6.100 FSUB (vector) on page D6-1357</a>
INS (vector, element)	Insert vector element from another vector element	<a href="#">D6.101 INS (vector, element) on page D6-1358</a>
INS (vector, general)	Insert vector element from general-purpose register	<a href="#">D6.102 INS (vector, general) on page D6-1359</a>
LD1 (vector, multiple structures)	Load multiple single-element structures to one, two, three, or four registers	<a href="#">D6.103 LD1 (vector, multiple structures) on page D6-1360</a>
LD1 (vector, single structure)	Load one single-element structure to one lane of one register	<a href="#">D6.104 LD1 (vector, single structure) on page D6-1363</a>
LD1R (vector)	Load one single-element structure and Replicate to all lanes (of one register)	<a href="#">D6.105 LD1R (vector) on page D6-1364</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
LD2 (vector, multiple structures)	Load multiple 2-element structures to two registers	<a href="#">D6.106 LD2 (vector, multiple structures) on page D6-1365</a>
LD2 (vector, single structure)	Load single 2-element structure to one lane of two registers	<a href="#">D6.107 LD2 (vector, single structure) on page D6-1366</a>
LD2R (vector)	Load single 2-element structure and Replicate to all lanes of two registers	<a href="#">D6.108 LD2R (vector) on page D6-1367</a>
LD3 (vector, multiple structures)	Load multiple 3-element structures to three registers	<a href="#">D6.109 LD3 (vector, multiple structures) on page D6-1368</a>
LD3 (vector, single structure)	Load single 3-element structure to one lane of three registers	<a href="#">D6.110 LD3 (vector, single structure) on page D6-1369</a>
LD3R (vector)	Load single 3-element structure and Replicate to all lanes of three registers	<a href="#">D6.111 LD3R (vector) on page D6-1371</a>
LD4 (vector, multiple structures)	Load multiple 4-element structures to four registers	<a href="#">D6.112 LD4 (vector, multiple structures) on page D6-1372</a>
LD4 (vector, single structure)	Load single 4-element structure to one lane of four registers	<a href="#">D6.113 LD4 (vector, single structure) on page D6-1373</a>
LD4R (vector)	Load single 4-element structure and Replicate to all lanes of four registers	<a href="#">D6.114 LD4R (vector) on page D6-1375</a>
MLA (vector, by element)	Multiply-Add to accumulator (vector, by element)	<a href="#">D6.115 MLA (vector, by element) on page D6-1376</a>
MLA (vector)	Multiply-Add to accumulator (vector)	<a href="#">D6.116 MLA (vector) on page D6-1377</a>
MLS (vector, by element)	Multiply-Subtract from accumulator (vector, by element)	<a href="#">D6.117 MLS (vector, by element) on page D6-1378</a>
MLS (vector)	Multiply-Subtract from accumulator (vector)	<a href="#">D6.118 MLS (vector) on page D6-1379</a>
MOV (vector, element)	Move vector element to another vector element	<a href="#">D6.119 MOV (vector, element) on page D6-1380</a>
MOV (vector, from general)	Move general-purpose register to a vector element	<a href="#">D6.120 MOV (vector, from general) on page D6-1381</a>
MOV (vector)	Move vector	<a href="#">D6.121 MOV (vector) on page D6-1382</a>
MOV (vector, to general)	Move vector element to general-purpose register	<a href="#">D6.122 MOV (vector, to general) on page D6-1383</a>
MOVI (vector)	Move Immediate (vector)	<a href="#">D6.123 MOVI (vector) on page D6-1384</a>
MUL (vector, by element)	Multiply (vector, by element)	<a href="#">D6.124 MUL (vector, by element) on page D6-1386</a>
MUL (vector)	Multiply (vector)	<a href="#">D6.125 MUL (vector) on page D6-1387</a>
MVN (vector)	Bitwise NOT (vector)	<a href="#">D6.126 MVN (vector) on page D6-1388</a>
MVNI (vector)	Move inverted Immediate (vector)	<a href="#">D6.127 MVNI (vector) on page D6-1389</a>
NEG (vector)	Negate (vector)	<a href="#">D6.128 NEG (vector) on page D6-1390</a>
NOT (vector)	Bitwise NOT (vector)	<a href="#">D6.129 NOT (vector) on page D6-1391</a>
ORN (vector)	Bitwise inclusive OR NOT (vector)	<a href="#">D6.130 ORN (vector) on page D6-1392</a>
ORR (vector, immediate)	Bitwise inclusive OR (vector, immediate)	<a href="#">D6.131 ORR (vector, immediate) on page D6-1393</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
ORR (vector, register)	Bitwise inclusive OR (vector, register)	<a href="#">D6.132 ORR (vector, register) on page D6-1394</a>
PMUL (vector)	Polynomial Multiply	<a href="#">D6.133 PMUL (vector) on page D6-1395</a>
PMULL, PMULL2 (vector)	Polynomial Multiply Long	<a href="#">D6.134 PMULL, PMULL2 (vector) on page D6-1396</a>
RADDHN, RADDHN2 (vector)	Rounding Add returning High Narrow	<a href="#">D6.135 RADDHN, RADDHN2 (vector) on page D6-1397</a>
RBIT (vector)	Reverse Bit order (vector)	<a href="#">D6.136 RBIT (vector) on page D6-1398</a>
REV16 (vector)	Reverse elements in 16-bit halfwords (vector)	<a href="#">D6.137 REV16 (vector) on page D6-1399</a>
REV32 (vector)	Reverse elements in 32-bit words (vector)	<a href="#">D6.138 REV32 (vector) on page D6-1400</a>
REV64 (vector)	Reverse elements in 64-bit doublewords (vector)	<a href="#">D6.139 REV64 (vector) on page D6-1401</a>
RSHRN, RSHRN2 (vector)	Rounding Shift Right Narrow (immediate)	<a href="#">D6.140 RSHRN, RSHRN2 (vector) on page D6-1402</a>
RSUBHN, RSUBHN2 (vector)	Rounding Subtract returning High Narrow	<a href="#">D6.141 RSUBHN, RSUBHN2 (vector) on page D6-1403</a>
SABA (vector)	Signed Absolute difference and Accumulate	<a href="#">D6.142 SABA (vector) on page D6-1404</a>
SABAL, SABAL2 (vector)	Signed Absolute difference and Accumulate Long	<a href="#">D6.143 SABAL, SABAL2 (vector) on page D6-1405</a>
SABD (vector)	Signed Absolute Difference	<a href="#">D6.144 SABD (vector) on page D6-1406</a>
SABDL, SABDL2 (vector)	Signed Absolute Difference Long	<a href="#">D6.145 SABDL, SABDL2 (vector) on page D6-1407</a>
SADALP (vector)	Signed Add and Accumulate Long Pairwise	<a href="#">D6.146 SADALP (vector) on page D6-1408</a>
SADDL, SADDL2 (vector)	Signed Add Long (vector)	<a href="#">D6.147 SADDL, SADDL2 (vector) on page D6-1409</a>
SADDLP (vector)	Signed Add Long Pairwise	<a href="#">D6.148 SADDLP (vector) on page D6-1410</a>
SADDLV (vector)	Signed Add Long across Vector	<a href="#">D6.149 SADDLV (vector) on page D6-1411</a>
SADDW, SADDW2 (vector)	Signed Add Wide	<a href="#">D6.150 SADDW, SADDW2 (vector) on page D6-1412</a>
SCVTF (vector, fixed-point)	Signed fixed-point Convert to Floating-point (vector)	<a href="#">D6.151 SCVTF (vector, fixed-point) on page D6-1413</a>
SCVTF (vector, integer)	Signed integer Convert to Floating-point (vector)	<a href="#">D6.152 SCVTF (vector, integer) on page D6-1414</a>
SDOT (vector, by element)	Dot Product signed arithmetic (vector, by element)	<a href="#">D6.153 SDOT (vector, by element) on page D6-1415</a>
SDOT (vector)	Dot Product signed arithmetic (vector)	<a href="#">D6.154 SDOT (vector) on page D6-1416</a>
SHADD (vector)	Signed Halving Add	<a href="#">D6.155 SHADD (vector) on page D6-1417</a>
SHL (vector)	Shift Left (immediate)	<a href="#">D6.156 SHL (vector) on page D6-1418</a>
SHLL, SHLL2 (vector)	Shift Left Long (by element size)	<a href="#">D6.157 SHLL, SHLL2 (vector) on page D6-1419</a>
SHRN, SHRN2 (vector)	Shift Right Narrow (immediate)	<a href="#">D6.158 SHRN, SHRN2 (vector) on page D6-1420</a>
SHSUB (vector)	Signed Halving Subtract	<a href="#">D6.159 SHSUB (vector) on page D6-1421</a>
SLI (vector)	Shift Left and Insert (immediate)	<a href="#">D6.160 SLI (vector) on page D6-1422</a>
SMAX (vector)	Signed Maximum (vector)	<a href="#">D6.161 SMAX (vector) on page D6-1423</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

Mnemonic	Brief description	See
SMAXP (vector)	Signed Maximum Pairwise	<a href="#">D6.162 SMAXP (vector) on page D6-1424</a>
SMAXV (vector)	Signed Maximum across Vector	<a href="#">D6.163 SMAXV (vector) on page D6-1425</a>
SMIN (vector)	Signed Minimum (vector)	<a href="#">D6.164 SMIN (vector) on page D6-1426</a>
SMINP (vector)	Signed Minimum Pairwise	<a href="#">D6.165 SMINP (vector) on page D6-1427</a>
SMINV (vector)	Signed Minimum across Vector	<a href="#">D6.166 SMINV (vector) on page D6-1428</a>
SMLAL, SMLAL2 (vector, by element)	Signed Multiply-Add Long (vector, by element)	<a href="#">D6.167 SMLAL, SMLAL2 (vector, by element) on page D6-1429</a>
SMLAL, SMLAL2 (vector)	Signed Multiply-Add Long (vector)	<a href="#">D6.168 SMLAL, SMLAL2 (vector) on page D6-1430</a>
SMLS L, SMLS L2 (vector, by element)	Signed Multiply-Subtract Long (vector, by element)	<a href="#">D6.169 SMLS L, SMLS L2 (vector, by element) on page D6-1431</a>
SMLS L, SMLS L2 (vector)	Signed Multiply-Subtract Long (vector)	<a href="#">D6.170 SMLS L, SMLS L2 (vector) on page D6-1432</a>
SMOV (vector)	Signed Move vector element to general-purpose register	<a href="#">D6.171 SMOV (vector) on page D6-1433</a>
SMULL, SMULL2 (vector, by element)	Signed Multiply Long (vector, by element)	<a href="#">D6.172 SMULL, SMULL2 (vector, by element) on page D6-1434</a>
SMULL, SMULL2 (vector)	Signed Multiply Long (vector)	<a href="#">D6.173 SMULL, SMULL2 (vector) on page D6-1435</a>
SQABS (vector)	Signed saturating Absolute value	<a href="#">D6.174 SQABS (vector) on page D6-1436</a>
SQADD (vector)	Signed saturating Add	<a href="#">D6.175 SQADD (vector) on page D6-1437</a>
SQDMLAL, SQDMLAL2 (vector, by element)	Signed saturating Doubling Multiply-Add Long (by element)	<a href="#">D6.176 SQDMLAL, SQDMLAL2 (vector, by element) on page D6-1438</a>
SQDMLAL, SQDMLAL2 (vector)	Signed saturating Doubling Multiply-Add Long	<a href="#">D6.177 SQDMLAL, SQDMLAL2 (vector) on page D6-1440</a>
SQDMLSL, SQDMLSL2 (vector, by element)	Signed saturating Doubling Multiply-Subtract Long (by element)	<a href="#">D6.178 SQDMLSL, SQDMLSL2 (vector, by element) on page D6-1441</a>
SQDMLSL, SQDMLSL2 (vector)	Signed saturating Doubling Multiply-Subtract Long	<a href="#">D6.179 SQDMLSL, SQDMLSL2 (vector) on page D6-1443</a>
SQDMULH (vector, by element)	Signed saturating Doubling Multiply returning High half (by element)	<a href="#">D6.180 SQDMULH (vector, by element) on page D6-1444</a>
SQDMULH (vector)	Signed saturating Doubling Multiply returning High half	<a href="#">D6.181 SQDMULH (vector) on page D6-1445</a>
SQDMULL, SQDMULL2 (vector, by element)	Signed saturating Doubling Multiply Long (by element)	<a href="#">D6.182 SQDMULL, SQDMULL2 (vector, by element) on page D6-1446</a>
SQDMULL, SQDMULL2 (vector)	Signed saturating Doubling Multiply Long	<a href="#">D6.183 SQDMULL, SQDMULL2 (vector) on page D6-1448</a>
SQNEG (vector)	Signed saturating Negate	<a href="#">D6.184 SQNEG (vector) on page D6-1449</a>
SQRDMLAH (vector, by element)	Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element)	<a href="#">D6.185 SQRDMLAH (vector, by element) on page D6-1450</a>
SQRDMLAH (vector)	Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector)	<a href="#">D6.186 SQRDMLAH (vector) on page D6-1451</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

<b>Mnemonic</b>	<b>Brief description</b>	<b>See</b>
SQRDMLSH (vector, by element)	Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element)	<a href="#">D6.187 SQRDMLSH (vector, by element) on page D6-1452</a>
SQRDMLSH (vector)	Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector)	<a href="#">D6.188 SQRDMLSH (vector) on page D6-1453</a>
SQRDMULH (vector, by element)	Signed saturating Rounding Doubling Multiply returning High half (by element)	<a href="#">D6.189 SQRDMULH (vector, by element) on page D6-1454</a>
SQRDMULH (vector)	Signed saturating Rounding Doubling Multiply returning High half	<a href="#">D6.190 SQRDMULH (vector) on page D6-1455</a>
SQRSHL (vector)	Signed saturating Rounding Shift Left (register)	<a href="#">D6.191 SQRSHL (vector) on page D6-1456</a>
SQRSHRN, SQRSHRN2 (vector)	Signed saturating Rounded Shift Right Narrow (immediate)	<a href="#">D6.192 SQRSHRN, SQRSHRN2 (vector) on page D6-1457</a>
SQRSHRUN, SQRSHRUN2 (vector)	Signed saturating Rounded Shift Right Unsigned Narrow (immediate)	<a href="#">D6.193 SQRSHRUN, SQRSHRUN2 (vector) on page D6-1458</a>
SQSHL (vector, immediate)	Signed saturating Shift Left (immediate)	<a href="#">D6.194 SQSHL (vector, immediate) on page D6-1459</a>
SQSHL (vector, register)	Signed saturating Shift Left (register)	<a href="#">D6.195 SQSHL (vector, register) on page D6-1460</a>
SQSHLU (vector)	Signed saturating Shift Left Unsigned (immediate)	<a href="#">D6.196 SQSHLU (vector) on page D6-1461</a>
SQSHRN, SQSHRN2 (vector)	Signed saturating Shift Right Narrow (immediate)	<a href="#">D6.197 SQSHRN, SQSHRN2 (vector) on page D6-1462</a>
SQSHRUN, SQSHRUN2 (vector)	Signed saturating Shift Right Unsigned Narrow (immediate)	<a href="#">D6.198 SQSHRUN, SQSHRUN2 (vector) on page D6-1463</a>
SQSUB (vector)	Signed saturating Subtract	<a href="#">D6.199 SQSUB (vector) on page D6-1464</a>
SQXTN, SQXTN2 (vector)	Signed saturating extract Narrow	<a href="#">D6.200 SQXTN, SQXTN2 (vector) on page D6-1465</a>
SQXTUN, SQXTUN2 (vector)	Signed saturating extract Unsigned Narrow	<a href="#">D6.201 SQXTUN, SQXTUN2 (vector) on page D6-1466</a>
SRHADD (vector)	Signed Rounding Halving Add	<a href="#">D6.202 SRHADD (vector) on page D6-1467</a>
SRI (vector)	Shift Right and Insert (immediate)	<a href="#">D6.203 SRI (vector) on page D6-1468</a>
SRSHL (vector)	Signed Rounding Shift Left (register)	<a href="#">D6.204 SRSHL (vector) on page D6-1469</a>
SRSHR (vector)	Signed Rounding Shift Right (immediate)	<a href="#">D6.205 SRSHR (vector) on page D6-1470</a>
SRSRA (vector)	Signed Rounding Shift Right and Accumulate (immediate)	<a href="#">D6.206 SRSRA (vector) on page D6-1471</a>
SSHLL (vector)	Signed Shift Left (register)	<a href="#">D6.207 SSHL (vector) on page D6-1472</a>
SSHLL, SSHLL2 (vector)	Signed Shift Left Long (immediate)	<a href="#">D6.208 SSHLL, SSHLL2 (vector) on page D6-1473</a>
SSHR (vector)	Signed Shift Right (immediate)	<a href="#">D6.209 SSHR (vector) on page D6-1474</a>
SSRA (vector)	Signed Shift Right and Accumulate (immediate)	<a href="#">D6.210 SSRA (vector) on page D6-1475</a>
SSUBL, SSUBL2 (vector)	Signed Subtract Long	<a href="#">D6.211 SSUBL, SSUBL2 (vector) on page D6-1476</a>
SSUBW, SSUBW2 (vector)	Signed Subtract Wide	<a href="#">D6.212 SSUBW, SSUBW2 (vector) on page D6-1477</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

Mnemonic	Brief description	See
ST1 (vector, multiple structures)	Store multiple single-element structures from one, two, three, or four registers	<a href="#">D6.213 ST1 (vector, multiple structures) on page D6-1478</a>
ST1 (vector, single structure)	Store a single-element structure from one lane of one register	<a href="#">D6.214 ST1 (vector, single structure) on page D6-1481</a>
ST2 (vector, multiple structures)	Store multiple 2-element structures from two registers	<a href="#">D6.215 ST2 (vector, multiple structures) on page D6-1482</a>
ST2 (vector, single structure)	Store single 2-element structure from one lane of two registers	<a href="#">D6.216 ST2 (vector, single structure) on page D6-1483</a>
ST3 (vector, multiple structures)	Store multiple 3-element structures from three registers	<a href="#">D6.217 ST3 (vector, multiple structures) on page D6-1484</a>
ST3 (vector, single structure)	Store single 3-element structure from one lane of three registers	<a href="#">D6.218 ST3 (vector, single structure) on page D6-1485</a>
ST4 (vector, multiple structures)	Store multiple 4-element structures from four registers	<a href="#">D6.219 ST4 (vector, multiple structures) on page D6-1487</a>
ST4 (vector, single structure)	Store single 4-element structure from one lane of four registers	<a href="#">D6.220 ST4 (vector, single structure) on page D6-1488</a>
SUB (vector)	Subtract (vector)	<a href="#">D6.221 SUB (vector) on page D6-1490</a>
SUBHN, SUBHN2 (vector)	Subtract returning High Narrow	<a href="#">D6.222 SUBHN, SUBHN2 (vector) on page D6-1491</a>
SUQADD (vector)	Signed saturating Accumulate of Unsigned value	<a href="#">D6.223 SUQADD (vector) on page D6-1492</a>
SXTL, SXTL2 (vector)	Signed extend Long	<a href="#">D6.224 SXTL, SXTL2 (vector) on page D6-1493</a>
TBL (vector)	Table vector Lookup	<a href="#">D6.225 TBL (vector) on page D6-1494</a>
TBX (vector)	Table vector lookup extension	<a href="#">D6.226 TBX (vector) on page D6-1495</a>
TRN1 (vector)	Transpose vectors (primary)	<a href="#">D6.227 TRN1 (vector) on page D6-1496</a>
TRN2 (vector)	Transpose vectors (secondary)	<a href="#">D6.228 TRN2 (vector) on page D6-1497</a>
UABA (vector)	Unsigned Absolute difference and Accumulate	<a href="#">D6.229 UABA (vector) on page D6-1498</a>
UABAL, UABAL2 (vector)	Unsigned Absolute difference and Accumulate Long	<a href="#">D6.230 UABAL, UABAL2 (vector) on page D6-1499</a>
UABD (vector)	Unsigned Absolute Difference (vector)	<a href="#">D6.231 UABD (vector) on page D6-1500</a>
UABDL, UABDL2 (vector)	Unsigned Absolute Difference Long	<a href="#">D6.232 UABDL, UABDL2 (vector) on page D6-1501</a>
UADALP (vector)	Unsigned Add and Accumulate Long Pairwise	<a href="#">D6.233 UADALP (vector) on page D6-1502</a>
UADDL, UADDL2 (vector)	Unsigned Add Long (vector)	<a href="#">D6.234 UADDL, UADDL2 (vector) on page D6-1503</a>
UADDLP (vector)	Unsigned Add Long Pairwise	<a href="#">D6.235 UADDLP (vector) on page D6-1504</a>
UADDLV (vector)	Unsigned sum Long across Vector	<a href="#">D6.236 UADDLV (vector) on page D6-1505</a>
UADDW, UADDW2 (vector)	Unsigned Add Wide	<a href="#">D6.237 UADDW, UADDW2 (vector) on page D6-1506</a>
UCVTF (vector, fixed-point)	Unsigned fixed-point Convert to Floating-point (vector)	<a href="#">D6.238 UCVTF (vector, fixed-point) on page D6-1507</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

Mnemonic	Brief description	See
UCVTF (vector, integer)	Unsigned integer Convert to Floating-point (vector)	<a href="#">D6.239 UCVTF (vector, integer) on page D6-1508</a>
UDOT (vector, by element)	Dot Product unsigned arithmetic (vector, by element)	<a href="#">D6.240 UDOT (vector, by element) on page D6-1509</a>
UDOT (vector)	Dot Product unsigned arithmetic (vector)	<a href="#">D6.241 UDOT (vector) on page D6-1510</a>
UHADD (vector)	Unsigned Halving Add	<a href="#">D6.242 UHADD (vector) on page D6-1511</a>
UHSUB (vector)	Unsigned Halving Subtract	<a href="#">D6.243 UHSUB (vector) on page D6-1512</a>
UMAX (vector)	Unsigned Maximum (vector)	<a href="#">D6.244 UMAX (vector) on page D6-1513</a>
UMAXP (vector)	Unsigned Maximum Pairwise	<a href="#">D6.245 UMAXP (vector) on page D6-1514</a>
UMAXV (vector)	Unsigned Maximum across Vector	<a href="#">D6.246 UMAXV (vector) on page D6-1515</a>
UMIN (vector)	Unsigned Minimum (vector)	<a href="#">D6.247 UMIN (vector) on page D6-1516</a>
UMINP (vector)	Unsigned Minimum Pairwise	<a href="#">D6.248 UMINP (vector) on page D6-1517</a>
UMINV (vector)	Unsigned Minimum across Vector	<a href="#">D6.249 UMINV (vector) on page D6-1518</a>
UMLAL, UMLAL2 (vector, by element)	Unsigned Multiply-Add Long (vector, by element)	<a href="#">D6.250 UMLAL, UMLAL2 (vector, by element) on page D6-1519</a>
UMLAL, UMLAL2 (vector)	Unsigned Multiply-Add Long (vector)	<a href="#">D6.251 UMLAL, UMLAL2 (vector) on page D6-1520</a>
UMLSL, UMLSL2 (vector, by element)	Unsigned Multiply-Subtract Long (vector, by element)	<a href="#">D6.252 UMLSL, UMLSL2 (vector, by element) on page D6-1521</a>
UMLSL, UMLSL2 (vector)	Unsigned Multiply-Subtract Long (vector)	<a href="#">D6.253 UMLSL, UMLSL2 (vector) on page D6-1522</a>
UMOV (vector)	Unsigned Move vector element to general-purpose register	<a href="#">D6.254 UMOV (vector) on page D6-1523</a>
UMULL, UMULL2 (vector, by element)	Unsigned Multiply Long (vector, by element)	<a href="#">D6.255 UMULL, UMULL2 (vector, by element) on page D6-1524</a>
UMULL, UMULL2 (vector)	Unsigned Multiply long (vector)	<a href="#">D6.256 UMULL, UMULL2 (vector) on page D6-1525</a>
UQADD (vector)	Unsigned saturating Add	<a href="#">D6.257 UQADD (vector) on page D6-1526</a>
UQRSHL (vector)	Unsigned saturating Rounding Shift Left (register)	<a href="#">D6.258 UQRSHL (vector) on page D6-1527</a>
UQRSHRN, UQRSHRN2 (vector)	Unsigned saturating Rounded Shift Right Narrow (immediate)	<a href="#">D6.259 UQRSHRN, UQRSHRN2 (vector) on page D6-1528</a>
UQSHL (vector, immediate)	Unsigned saturating Shift Left (immediate)	<a href="#">D6.260 UQSHL (vector, immediate) on page D6-1529</a>
UQSHL (vector, register)	Unsigned saturating Shift Left (register)	<a href="#">D6.261 UQSHL (vector, register) on page D6-1530</a>
UQSHRN, UQSHRN2 (vector)	Unsigned saturating Shift Right Narrow (immediate)	<a href="#">D6.262 UQSHRN, UQSHRN2 (vector) on page D6-1531</a>
UQSUB (vector)	Unsigned saturating Subtract	<a href="#">D6.263 UQSUB (vector) on page D6-1533</a>
UQXTN, UQXTN2 (vector)	Unsigned saturating extract Narrow	<a href="#">D6.264 UQXTN, UQXTN2 (vector) on page D6-1534</a>
URECPE (vector)	Unsigned Reciprocal Estimate	<a href="#">D6.265 URECPE (vector) on page D6-1535</a>
URHADD (vector)	Unsigned Rounding Halving Add	<a href="#">D6.266 URHADD (vector) on page D6-1536</a>

**Table D6-1 Summary of A64 SIMD Vector instructions (continued)**

Mnemonic	Brief description	See
URSHL (vector)	Unsigned Rounding Shift Left (register)	<a href="#">D6.267 URSHL (vector) on page D6-1537</a>
URSHR (vector)	Unsigned Rounding Shift Right (immediate)	<a href="#">D6.268 URSHR (vector) on page D6-1538</a>
URSQRTE (vector)	Unsigned Reciprocal Square Root Estimate	<a href="#">D6.269 URSQRTE (vector) on page D6-1539</a>
URSRA (vector)	Unsigned Rounding Shift Right and Accumulate (immediate)	<a href="#">D6.270 URSRA (vector) on page D6-1540</a>
USHL (vector)	Unsigned Shift Left (register)	<a href="#">D6.271 USHL (vector) on page D6-1541</a>
USHLL, USHLL2 (vector)	Unsigned Shift Left Long (immediate)	<a href="#">D6.272 USHLL, USHLL2 (vector) on page D6-1542</a>
USHR (vector)	Unsigned Shift Right (immediate)	<a href="#">D6.273 USHR (vector) on page D6-1543</a>
USQADD (vector)	Unsigned saturating Accumulate of Signed value	<a href="#">D6.274 USQADD (vector) on page D6-1544</a>
USRA (vector)	Unsigned Shift Right and Accumulate (immediate)	<a href="#">D6.275 USRA (vector) on page D6-1545</a>
USUBL, USUBL2 (vector)	Unsigned Subtract Long	<a href="#">D6.276 USUBL, USUBL2 (vector) on page D6-1546</a>
USUBW, USUBW2 (vector)	Unsigned Subtract Wide	<a href="#">D6.277 USUBW, USUBW2 (vector) on page D6-1547</a>
UXTL, UXTL2 (vector)	Unsigned extend Long	<a href="#">D6.278 UXTL, UXTL2 (vector) on page D6-1548</a>
UZP1 (vector)	Unzip vectors (primary)	<a href="#">D6.279 UZP1 (vector) on page D6-1549</a>
UZP2 (vector)	Unzip vectors (secondary)	<a href="#">D6.280 UZP2 (vector) on page D6-1550</a>
XTN, XTN2 (vector)	Extract Narrow	<a href="#">D6.281 XTN, XTN2 (vector) on page D6-1551</a>
ZIP1 (vector)	Zip vectors (primary)	<a href="#">D6.282 ZIP1 (vector) on page D6-1552</a>
ZIP2 (vector)	Zip vectors (secondary)	<a href="#">D6.283 ZIP2 (vector) on page D6-1553</a>

## D6.2 ABS (vector)

Absolute value (vector).

### Syntax

ABS  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD and FP register, puts the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.3 ADD (vector)

Add (vector).

### Syntax

ADD  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Add (vector). This instruction adds corresponding elements in the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and FP register.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.4 ADDHN, ADDHN2 (vector)

Add returning High Narrow.

### Syntax

$\text{ADDHN}\{2\} \text{ } Vd.Tb, \text{ } Vn.Ta, \text{ } Vm.Ta$

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Add returning High Narrow. This instruction adds each vector element in the first source SIMD and FP register to the corresponding vector element in the second source SIMD and FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register.

The results are truncated. For rounded results, see [D6.135 RADDHN, RADDHN2 \(vector\) on page D6-1397](#).

The ADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the ADDHN2 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.

The following table shows the valid specifier combinations:

**Table D6-2 ADDHN, ADDHN2 (Vector) specifier combinations**

<Q>	Tb	Ta
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.5 ADDP (vector)

Add Pairwise (vector).

### Syntax

ADDP  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.6 ADDV (vector)

Add across Vector.

### Syntax

ADDV *Vd*, *Vn.T*

Where:

*V*

Is the destination width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Add across Vector. This instruction adds every vector element in the source SIMD and FP register together, and writes the scalar result to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-3 ADDV (Vector) specifier combinations**

<i>V</i>	<i>T</i>
B	8B
B	16B
H	4H
H	8H
S	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.7 AND (vector)

Bitwise AND (vector).

### Syntax

AND  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be either 8B or 16B.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.8 BIC (vector, immediate)

Bitwise bit Clear (vector, immediate).

### Syntax

BIC *Vd.T*, #*imm8{*, LSL #*amount*} ; 16-bit

BIC *Vd.T*, #*imm8{*, LSL #*amount*} ; 32-bit

Where:

*T*

Is an arrangement specifier:

#### 16-bit

Can be one of 4H or 8H.

#### 32-bit

Can be one of 2S or 4S.

*amount*

Is the shift amount:

#### 16-bit

Can be one of 0 or 8.

#### 32-bit

Can be one of 0, 8, 16 or 24.

Defaults to zero if LSL is omitted.

*Vd*

Is the name of the SIMD and FP register.

*imm8*

Is an 8-bit immediate.

### Usage

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD and 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 and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.9 BIC (vector, register)

Bitwise bit Clear (vector, register).

### Syntax

BIC  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be either 8B or 16B.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD and FP register and the complement of the second source SIMD and FP register, and writes the result to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.10 BIF (vector)

Bitwise Insert if False.

### Syntax

BIF  $Vd.T, Vn.T,Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be either 8B or 16B.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD and FP register into the destination SIMD and FP register if the corresponding bit of the second source SIMD and FP register is 0, otherwise leaves the bit in the destination register unchanged.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.11 BIT (vector)

Bitwise Insert if True.

### Syntax

BIT  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be either 8B or 16B.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD and FP register into the SIMD and FP destination register if the corresponding bit of the second source SIMD and FP register is 1, otherwise leaves the bit in the destination register unchanged.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.12 BSL (vector)

Bitwise Select.

### Syntax

**BSL Vd.T, Vn.T,Vm.T**

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be either 8B or 16B.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Bitwise Select. This instruction sets each bit in the destination SIMD and FP register to the corresponding bit from the first source SIMD and FP register when the original destination bit was 1, otherwise from the second source SIMD and 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.

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.13 CLS (vector)

Count Leading Sign bits (vector).

### Syntax

CLS  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register. The count does not include the most significant bit itself.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.14 CLZ (vector)

Count Leading Zero bits (vector).

### Syntax

CLZ *Vd.T, Vn.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.15 CMEQ (vector, register)

Compare bitwise Equal (vector).

### Syntax

CMEQ  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD and FP register with the corresponding vector element from the second source SIMD and FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.16 CMEQ (vector, zero)

Compare bitwise Equal to zero (vector).

### Syntax

CMEQ *Vd.T, Vn.T, #0*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.17 CMGE (vector, register)

Compare signed Greater than or Equal (vector).

### Syntax

CMGE  $Vd.T$ ,  $Vn.T$ ,  $Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.18 CMGE (vector, zero)

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

### Syntax

CMGE *Vd.T, Vn.T, #0*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.19 CMGT (vector, register)

Compare signed Greater than (vector).

### Syntax

CMGT  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.20 CMGT (vector, zero)

Compare signed Greater than zero (vector).

### Syntax

CMGT *Vd.T, Vn.T, #0*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.21 CMHI (vector, register)

Compare unsigned Higher (vector).

### Syntax

CMHI  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.22 CMHS (vector, register)

Compare unsigned Higher or Same (vector).

### Syntax

CMHS  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.23 CMLE (vector, zero)

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

### Syntax

CMLE *Vd.T, Vn.T, #0*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.24 CMLT (vector, zero)

Compare signed Less than zero (vector).

### Syntax

CMLT *Vd.T, Vn.T, #0*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.25 CMTST (vector)

Compare bitwise Test bits nonzero (vector).

### Syntax

CMTST  $Vd.T$ ,  $Vn.T$ ,  $Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD and FP register, performs an AND with the corresponding vector element in the second source SIMD and FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.26 CNT (vector)

Population Count per byte.

### Syntax

CNT  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be either 8B or 16B.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.27 DUP (vector, element)

vector.

### Syntax

DUP *Vd.T, Vn.Ts[index]*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Ts*

Is an element size specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*index*

Is the element index, in the range shown in Usage.

### Usage

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD and FP register into a scalar or each element in a vector, and writes the result to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-4 DUP (Vector) specifier combinations**

<i>T</i>	<i>Ts</i>	<i>index</i>
8B	B	0 to 15
16B	B	0 to 15
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

### Related reference

[D5.1 A64 SIMD scalar instructions in alphabetical order](#) on page D5-1110

## D6.28 DUP (vector, general)

Duplicate general-purpose register to vector.

### Syntax

DUP *Vd.T, Rn*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*R*

Is the width specifier for the general-purpose source register, and can be either W or X.

*n*

Is the number [0-30] of the general-purpose source register or ZR (31).

### Usage

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-5 DUP (Vector) specifier combinations**

<i>T</i>	<i>R</i>
8B	W
16B	W
4H	W
8H	W
2S	W
4S	W
2D	X

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.29 EOR (vector)

Bitwise Exclusive OR (vector).

### Syntax

EOR  $Vd.T$ ,  $Vn.T$ ,  $Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be either 8B or 16B.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD and FP registers, and places the result in the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.30 EXT (vector)

Extract vector from pair of vectors.

### Syntax

`EXT Vd.T, Vn.T, Vm.T, #index`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be either 8B or 16B.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

**index**

Is the lowest numbered byte element to be extracted in the range shown in Usage.

### Usage

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD and FP register and the highest vector elements from the first source SIMD and FP register, concatenates the results into a vector, and writes the vector to the destination SIMD and 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.

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.

The following table shows the valid specifier combinations:

**Table D6-6 EXT (Vector) specifier combinations**

<b>T</b>	<b>index</b>
8B	0 to 7
16B	0 to 15

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.31 FABD (vector)

Floating-point Absolute Difference (vector).

### Syntax

FABD *Vd.T, Vn.T, Vm.T ; Vector half precision*

FABD *Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision*

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register

*Vm*

Is the name of the second SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD and FP register, from the corresponding floating-point values in the elements of the first source SIMD and FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.32 FABS (vector)

Floating-point Absolute value (vector).

### Syntax

FABS *Vd.T, Vn.T ; Half-precision*  
FABS *Vd.T, Vn.T ; Single-precision and double-precision*

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD and FP register, writes the result to a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.33 FACGE (vector)

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

### Syntax

```
FACGE Vd.T, Vn.T, Vm.T ; Vector half precision  
FACGE Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision
```

Where:

**Vd**

Is the name of the SIMD and FP destination register

**T**

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register

**Vm**

Is the name of the second SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD and FP register with the absolute value of the corresponding floating-point value in the second source SIMD and FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.34 FACGT (vector)

Floating-point Absolute Compare Greater than (vector).

### Syntax

`FACGT Vd.T, Vn.T, Vm.T ; Vector half precision`

`FACGT Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision`

Where:

**Vd**

Is the name of the SIMD and FP destination register

**T**

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register

**Vm**

Is the name of the second SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD and FP register with the absolute value of the corresponding vector element in the second source SIMD and FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.35 FADD (vector)

Floating-point Add (vector).

### Syntax

FADD *Vd.T, Vn.T, Vm.T* ; Half-precision

FADD *Vd.T, Vn.T, Vm.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD and FP registers, writes the result into a vector, and writes the vector to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.36 FADDP (vector)

Floating-point Add Pairwise (vector).

### Syntax

FADDP *Vd.T, Vn.T,Vm.T* ; Half-precision

FADDP *Vd.T, Vn.T,Vm.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.37 FCADD (vector)

Floating-point Complex Add.

### Syntax

FCADD *Vd.T, Vn.T, Vm.T, #rotate*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 4H, 8H, 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

*rotate*

Is the rotation, and can be either 90 or 270.

### Architectures supported (vector)

Supported in Armv8.3-A architecture and later.

### Usage

Floating-point Complex Add.

This instruction adds two source complex numbers from the *Vm* and the *Vn* vector registers and places the resulting complex number in the destination *Vd* vector register. The number of complex numbers that can be stored in the *Vm*, the *Vn*, and the *Vd* registers is calculated as the vector register size divided by the length of each complex number. These lengths are 16 for half-precision, 32 for single-precision, and 64 for double-precision. Each complex number is represented in a SIMD&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

One of the two vector elements that are read from each of the numbers in the *Vm* source SIMD and FP register can be optionally negated based on the rotation value:

- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.38 FCMEQ (vector, register)

Floating-point Compare Equal (vector).

### Syntax

FCMEQ *Vd.T, Vn.T,Vm.T* ; Vector half precision

FCMEQ *Vd.T, Vn.T,Vm.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD and FP register, with the corresponding floating-point value from the second source SIMD and FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.39 FCMEQ (vector, zero)

Floating-point Compare Equal to zero (vector).

### Syntax

FCMEQ *Vd.T, Vn.T, #0.0 ; Vector half precision*

FCMEQ *Vd.T, Vn.T, #0.0 ; Vector single-precision and double-precision*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.40 FCMGE (vector, register)

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

### Syntax

FCMGE *Vd.T, Vn.T,Vm.T* ; Vector half precision

FCMGE *Vd.T, Vn.T,Vm.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD and FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD and FP register sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.41 FCMGE (vector, zero)

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

### Syntax

FCMGE *Vd.T, Vn.T, #0.0 ; Vector half precision*

FCMGE *Vd.T, Vn.T, #0.0 ; Vector single-precision and double-precision*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.42 FCMGT (vector, register)

Floating-point Compare Greater than (vector).

### Syntax

FCMGT *Vd.T, Vn.T,Vm.T* ; Vector half precision

FCMGT *Vd.T, Vn.T,Vm.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD and FP register and if the value is greater than the corresponding floating-point value in the second source SIMD and FP register sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.43 FCMGT (vector, zero)

Floating-point Compare Greater than zero (vector).

### Syntax

FCMGT *Vd.T, Vn.T, #0.0 ; Vector half precision*

FCMGT *Vd.T, Vn.T, #0.0 ; Vector single-precision and double-precision*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.44 FCMLA (vector)

Floating-point Complex Multiply Accumulate.

### Syntax

FCMLA *Vd.T, Vn.T, Vm.T, #rotate*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 4H, 8H, 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

*rotate*

Is the rotation, and can be one of 0, 90, 180 or 270.

### Architectures supported (vector)

Supported in Armv8.3-A architecture and later.

### Usage

This instruction multiplies the two source complex numbers from the *Vm* and the *Vn* vector registers and adds the result to the corresponding complex number in the destination *Vd* vector register. The number of complex numbers that can be stored in the *Vm*, the *Vn*, and the *Vd* registers is calculated as the vector register size divided by the length of each complex number. These lengths are 16 for half-precision, 32 for single-precision, and 64 for double-precision. Each complex number is represented in a SIMD&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

None, one, or both of the two vector elements that are read from each of the numbers in the *Vm* source SIMD and FP register can be negated based on the rotation value:

- If the rotation is 0, none of the vector elements are negated.
- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 180, both vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.45 FCMLE (vector, zero)

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

### Syntax

FCMLE *Vd.T, Vn.T, #0.0 ; Vector half precision*

FCMLE *Vd.T, Vn.T, #0.0 ; Vector single-precision and double-precision*

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.46 FCMLT (vector, zero)

Floating-point Compare Less than zero (vector).

### Syntax

FCMLT *Vd.T, Vn.T, #0.0 ; Vector half precision*

FCMLT *Vd.T, Vn.T, #0.0 ; Vector single-precision and double-precision*

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.47 FCVTAS (vector)

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

### Syntax

FCVTAS *Vd.T, Vn.T* ; Vector half precision

FCVTAS *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.48 FCVTAU (vector)

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

### Syntax

FCVTAU *Vd.T, Vn.T* ; Vector half precision

FCVTAU *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.49 FCVTL, FCVTL2 (vector)

Floating-point Convert to higher precision Long (vector).

### Syntax

`FCVTL{2} Vd.Ta, Vn.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD and FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD and FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the FCVTL2 variant operates on the elements in the top 64 bits of the source 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

The following table shows the valid specifier combinations:

**Table D6-7 FCVTL, FCVTL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.50 FCVTMS (vector)

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

### Syntax

FCVTMS *Vd.T, Vn.T* ; Vector half precision

FCVTMS *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.51 FCVTMU (vector)

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

### Syntax

FCVTMU *Vd.T, Vn.T* ; Vector half precision

FCVTMU *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.52 FCVTN, FCVTN2 (vector)

Floating-point Convert to lower precision Narrow (vector).

### Syntax

`FCVTN{2} Vd.Tb, Vn.Ta`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

### Usage

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD and FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

The FCVTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-8 FCVTN, FCVTN2 (Vector) specifier combinations**

<Q>	Tb	Ta
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.53 FCVTNS (vector)

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

### Syntax

FCVTNS *Vd.T, Vn.T* ; Vector half precision

FCVTNS *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.54 FCVTNU (vector)

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

### Syntax

FCVTNU *Vd.T, Vn.T* ; Vector half precision

FCVTNU *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.55 FCVTPS (vector)

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

### Syntax

FCVTPS *Vd.T, Vn.T* ; Vector half precision

FCVTPS *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.56 FCVTPU (vector)

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

### Syntax

FCVTPU *Vd.T, Vn.T* ; Vector half precision

FCVTPU *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.57 FCVTXN, FCVTXN2 (vector)

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

### Syntax

`FCVTXN{2} Vd.Tb, Vn.Ta`

Where:

**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. See `<Q>` in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be either 2S or 4S.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, 2D.

### Usage

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD and FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD and FP register.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-9 FCVTXN{2} (Vector) specifier combinations**

<code>&lt;Q&gt;</code>	<b>Tb</b>	<b>Ta</b>
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.58 FCVTZS (vector, fixed-point)

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

### Syntax

`FCVTZS Vd.T, Vn.T, #fbits`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**fbits**

Is the number of fractional bits, in the range 1 to the element width.

### Usage

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-10 FCVTZS (Vector) specifier combinations**

<b>T</b>	<b>fbits</b>
4H	
8H	
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.59 FCVTZS (vector, integer)

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

### Syntax

FCVTZS *Vd.T, Vn.T* ; Vector half precision

FCVTZS *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.60 FCVTZU (vector, fixed-point)

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

### Syntax

FCVTZU *Vd.T, Vn.T, #fbits*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*fbits*

Is the number of fractional bits, in the range 1 to the element width.

### Usage

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-11 FCVTZU (Vector) specifier combinations**

<i>T</i>	<i>fbits</i>
4H	
8H	
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.61 FCVTZU (vector, integer)

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

### Syntax

FCVTZU *Vd.T, Vn.T* ; Vector half precision

FCVTZU *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.62 FDIV (vector)

Floating-point Divide (vector).

### Syntax

FDIV *Vd.T, Vn.T, Vm.T* ; Half-precision

FDIV *Vd.T, Vn.T, Vm.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD and FP register, by the floating-point values in the corresponding elements in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.63 FMAX (vector)

Floating-point Maximum (vector).

### Syntax

FMAX *Vd.T, Vn.T, Vm.T* ; Half-precision

FMAX *Vd.T, Vn.T, Vm.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD and FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.64 FMAXNM (vector)

Floating-point Maximum Number (vector).

### Syntax

FMAXNM *Vd.T, Vn.T, Vm.T ; Half-precision*

FMAXNM *Vd.T, Vn.T, Vm.T ; Single-precision and double-precision*

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD and FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD and 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 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.65 FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector).

### Syntax

FMAXNMP  $Vd.T, Vn.T, Vm.T$  ; Half-precision

FMAXNMP  $Vd.T, Vn.T, Vm.T$  ; Single-precision and double-precision

Where:

$T$

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

$Vd$

Is the name of the SIMD and FP destination register.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD and 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 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.66 FMAXNMV (vector)

Floating-point Maximum Number across Vector.

### Syntax

FMAXNMV *Vd*, *Vn.T* ; Half-precision  
FMAXNMV *Vd*, *Vn.T* ; Single-precision and double-precision

Where:

**v**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Must be S.

**T**

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Must be 4S.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the largest of the values as a scalar to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.67 FMAXP (vector)

Floating-point Maximum Pairwise (vector).

### Syntax

FMAXP *Vd.T, Vn.T,Vm.T* ; Half-precision

FMAXP *Vd.T, Vn.T,Vm.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.68 FMAXV (vector)

Floating-point Maximum across Vector.

### Syntax

FMAXV *Vd*, *Vn.T* ; Half-precision  
FMAXV *Vd*, *Vn.T* ; Single-precision and double-precision

Where:

**v**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Must be S.

**T**

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Must be 4S.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the largest of the values as a scalar to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.69 FMIN (vector)

Floating-point minimum (vector).

### Syntax

FMIN  $Vd.T, Vn.T, Vm.T$  ; Half-precision

FMIN  $Vd.T, Vn.T, Vm.T$  ; Single-precision and double-precision

Where:

$T$

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

$Vd$

Is the name of the SIMD and FP destination register.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD and FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.70 FMINNM (vector)

Floating-point Minimum Number (vector).

### Syntax

`FMINNM Vd.T, Vn.T, Vm.T ; Half-precision`

`FMINNM Vd.T, Vn.T, Vm.T ; Single-precision and double-precision`

Where:

`T`

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

`Vd`

Is the name of the SIMD and FP destination register.

`Vn`

Is the name of the first SIMD and FP source register.

`Vm`

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD and FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD and 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 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.71 FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector).

### Syntax

FMINNMP *Vd.T, Vn.T,Vm.T* ; Half-precision

FMINNMP *Vd.T, Vn.T,Vm.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD and 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 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.72 FMINNMV (vector)

Floating-point Minimum Number across Vector.

### Syntax

`FMINNMV Vd, Vn.T ; Half-precision`  
`FMINNMV Vd, Vn.T ; Single-precision and double-precision`

Where:

**v**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Must be S.

**T**

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Must be 4S.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the smallest of the values as a scalar to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.73 FMINP (vector)

Floating-point Minimum Pairwise (vector).

### Syntax

FMINP *Vd.T, Vn.T,Vm.T* ; Half-precision

FMINP *Vd.T, Vn.T,Vm.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.74 FMINV (vector)

Floating-point Minimum across Vector.

### Syntax

FMINV *Vd*, *Vn.T* ; Half-precision  
FMINV *Vd*, *Vn.T* ; Single-precision and double-precision

Where:

**v**

Is the destination width specifier:

#### Half-precision

Must be H.

#### Single-precision and double-precision

Must be S.

**T**

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Must be 4S.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the smallest of the values as a scalar to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.75 FMLA (vector, by element)

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

### Syntax

FMLA *Vd.T, Vn.T, Vm.Ts[index]* ; Vector, half-precision

FMLA *Vd.T, Vn.T, Vm.Ts[index]* ; Vector, single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector, half-precision

Can be one of 4H or 8H.

#### Vector, single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Ts*

Is an element size specifier:

#### Vector, half-precision

Must be H.

#### Vector, single-precision and double-precision

Can be one of S or D.

*index*

Is the element index:

#### Vector, half-precision

Must be H:L:M.

#### Vector, single-precision and double-precision

Can be one of H:L or H.

*Vm*

Is the name of the second SIMD and FP source register in the range 0 to 31.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the results in the vector elements of the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-12 FMLA (Vector, single-precision and double-precision) specifier combinations**

<i>T</i>	<i>Ts</i>	<i>index</i>
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

**Related reference**

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

**Related information**

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.76 FMLA (vector)

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

### Syntax

FMLA  $Vd.T, Vn.T, Vm.T$

Where:

$T$

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

$Vd$

Is the name of the SIMD and FP destination register.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD and FP registers, adds the product to the corresponding vector element of the destination SIMD and FP register, and writes the result to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.77 FMLAL, (vector)

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

### Syntax

FMLAL *Vd.Ta, Vn.Tb, Vm.Tb ; FMLAL*

FMLAL2 *Vd.Ta, Vn.Tb, Vm.Tb ; FMLAL2*

Where:

*Vd*

Is the name of the SIMD and FP destination register

*Ta*

Is an arrangement specifier, and can be one of 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register

*Tb*

Is an arrangement specifier, and can be one of 2H or 4H.

*Vm*

Is the name of the second SIMD and FP source register

### Architectures supported (vector)

Supported in Armv8.2 and later.

### Usage

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD and FP registers, and accumulates the product to the corresponding vector element of the destination SIMD and FP register. The instruction does not round the result of the multiply before the accumulation.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

---

#### Note

---

ID\_AA64ISAR0\_EL1.FHM indicates whether this instruction is supported. See *ID\_AA64ISAR0\_EL1* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

#### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.78 FMLS (vector, by element)

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

### Syntax

FMLS *Vd.T, Vn.T, Vm.Ts[index]* ; Vector, half-precision

FMLS *Vd.T, Vn.T, Vm.Ts[index]* ; Vector, single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector, half-precision

Can be one of 4H or 8H.

#### Vector, single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Ts*

Is an element size specifier:

#### Vector, half-precision

Must be H.

#### Vector, single-precision and double-precision

Can be one of S or D.

*index*

Is the element index:

#### Vector, half-precision

Must be H:L:M.

#### Vector, single-precision and double-precision

Can be one of H:L or H.

*Vm*

Is the name of the second SIMD and FP source register in the range 0 to 31.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and subtracts the results from the vector elements of the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-13 FMLS (Vector, single-precision and double-precision) specifier combinations**

<i>T</i>	<i>Ts</i>	<i>index</i>
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

**Related reference**

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

**Related information**

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.79 FMLS (vector)

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

### Syntax

**FMLS *Vd.T, Vn.T, Vm.T***

Where:

**T**

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

**Vd**

Is the name of the SIMD and FP destination register.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD and FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD and FP register, and writes the result to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.80 FMLS<sub>L</sub>, (vector)

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

### Syntax

FMLS<sub>L</sub> *Vd.Ta, Vn.Tb, Vm.Tb ; FMLS*

FMLS<sub>L2</sub> *Vd.Ta, Vn.Tb, Vm.Tb ; FMLS<sub>L2</sub>*

Where:

*Vd*

Is the name of the SIMD and FP destination register

*Ta*

Is an arrangement specifier, and can be one of 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register

*Tb*

Is an arrangement specifier, and can be one of 2H or 4H.

*Vm*

Is the name of the second SIMD and FP source register

### Architectures supported (vector)

Supported in Armv8.2 and later.

### Usage

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD and FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD and FP register. The instruction does not round the result of the multiply before the accumulation.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

---

#### Note

---

ID\_AA64ISAR0\_EL1.FHM indicates whether this instruction is supported. See *ID\_AA64ISAR0\_EL1* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

---

#### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

#### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.81 FMOV (vector, immediate)

Floating-point move immediate (vector).

### Syntax

```
FMOV Vd.T, #imm ; Half-precision  
FMOV Vd.T, #imm ; Single-precision  
FMOV Vd.2D, #imm ; Double-precision
```

Where:

**Vd**

The value depends on the instruction variant:

#### Half-precision

Is the name of the SIMD and FP destination register

#### Single-precision

Is the name of the SIMD and FP destination register

#### Double-precision

Is the name of the SIMD and FP destination register

**T**

Is an arrangement specifier:

#### Half-precision

Can be one of **4H** or **8H**.

#### Single-precision

Can be one of **2S** or **4S**.

**imm**

The value depends on the instruction variant:

#### Half-precision

Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision. For details of the range of constants available and the encoding of **imm**, see *Modified immediate constants in A64 floating-point instructions* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

#### Single-precision

Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision. For details of the range of constants available and the encoding of **imm**, see *Modified immediate constants in A64 floating-point instructions* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

#### Double-precision

Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision. For details of the range of constants available and the encoding of **imm**, see *Modified immediate constants in A64 floating-point instructions* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

## Usage

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.82 FMUL (vector, by element)

Floating-point Multiply (by element).

### Syntax

FMUL *Vd.T, Vn.T, Vm.Ts[index]* ; Vector, half-precision

FMUL *Vd.T, Vn.T, Vm.Ts[index]* ; Vector, single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector, half-precision

Can be one of 4H or 8H.

#### Vector, single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Ts*

Is an element size specifier:

#### Vector, half-precision

Must be H.

#### Vector, single-precision and double-precision

Can be one of S or D.

*index*

Is the element index:

#### Vector, half-precision

Must be H:L:M.

#### Vector, single-precision and double-precision

Can be one of H:L or H.

*Vm*

Is the name of the second SIMD and FP source register in the range 0 to 31.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-14 FMUL (Vector, single-precision and double-precision) specifier combinations**

<i>T</i>	<i>Ts</i>	<i>index</i>
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

**Related reference**

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

**Related information**

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.83 FMUL (vector)

Floating-point Multiply (vector).

### Syntax

FMUL *Vd.T, Vn.T, Vm.T*

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD and FP registers, places the result in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.84 FMULX (vector, by element)

Floating-point Multiply extended (by element).

### Syntax

FMULX *Vd.T, Vn.T, Vm.Ts[index]* ; Vector, half-precision

FMULX *Vd.T, Vn.T, Vm.Ts[index]* ; Vector, single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector, half-precision

Can be one of 4H or 8H.

#### Vector, single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Ts*

Is an element size specifier:

#### Vector, half-precision

Must be H.

#### Vector, single-precision and double-precision

Can be one of S or D.

*index*

Is the element index:

#### Vector, half-precision

Must be H:L:M.

#### Vector, single-precision and double-precision

Can be one of H:L or H.

*Vm*

Is the name of the second SIMD and FP source register in the range 0 to 31.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD and FP register by the specified floating-point value in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

Before each multiplication, a check is performed for whether one value is infinite and the other is zero. In this case, if only one of the values is negative, the result is 2.0, otherwise the result is -2.0.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-15 FMULX (Vector, single-precision and double-precision) specifier combinations**

<i>T</i>	<i>Ts</i>	<i>index</i>
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

**Related information**

*Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*

## D6.85 FMULX (vector)

Floating-point Multiply extended.

### Syntax

FMULX *Vd.T, Vn.T, Vm.T* ; Vector half precision

FMULX *Vd.T, Vn.T, Vm.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD and FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD and FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.86 FNEG (vector)

Floating-point Negate (vector).

### Syntax

FNEG *Vd.T, Vn.T* ; Half-precision  
FNEG *Vd.T, Vn.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD and FP register, writes the result to a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.87 FRECPE (vector)

Floating-point Reciprocal Estimate.

### Syntax

`FRECPE Vd.T, Vn.T ; Vector half precision`

`FRECPE Vd.T, Vn.T ; Vector single-precision and double-precision`

Where:

**Vd**

Is the name of the SIMD and FP destination register

**T**

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

**Vn**

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.88 FRECPS (vector)

Floating-point Reciprocal Step.

### Syntax

`FRECPS Vd.T, Vn.T, Vm.T ; Vector half precision`

`FRECPS Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision`

Where:

**Vd**

Is the name of the SIMD and FP destination register

**T**

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register

**Vm**

Is the name of the second SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD and FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.89 FRECPX (vector)

Floating-point Reciprocal exponent (scalar).

### Syntax

```
FRECPX Hd, Hn ; Half-precision  
FRECPX Vd, Vn ; Single-precision and double-precision
```

Where:

- Hd** Is the 16-bit name of the SIMD and FP destination register.
- Hn** Is the 16-bit name of the SIMD and FP source register.
- V** Is a width specifier, and can be either S or D.
- d** Is the number of the SIMD and FP destination register.
- n** Is the number of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.90 FRINTA (vector)

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

### Syntax

FRINTA *Vd.T, Vn.T* ; Half-precision

FRINTA *Vd.T, Vn.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.91 FRINTI (vector)

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

### Syntax

```
FRINTI Vd.T, Vn.T ; Half-precision  
FRINTI Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.92 FRINTM (vector)

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

### Syntax

FRINTM *Vd.T, Vn.T* ; Half-precision

FRINTM *Vd.T, Vn.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.93 FRINTN (vector)

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

### Syntax

FRINTN *Vd.T, Vn.T* ; Half-precision

FRINTN *Vd.T, Vn.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.94 FRINTP (vector)

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

### Syntax

```
FRINTP Vd.T, Vn.T ; Half-precision  
FRINTP Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.95 FRINTX (vector)

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

### Syntax

FRINTX *Vd.T, Vn.T* ; Half-precision

FRINTX *Vd.T, Vn.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.96 FRINTZ (vector)

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

### Syntax

```
FRINTZ Vd.T, Vn.T ; Half-precision  
FRINTZ Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.97 FRSQRTE (vector)

Floating-point Reciprocal Square Root Estimate.

### Syntax

FRSQRTE *Vd.T, Vn.T* ; Vector half precision

FRSQRTE *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.98 FRSQRTS (vector)

Floating-point Reciprocal Square Root Step.

### Syntax

FRSQRTS *Vd.T, Vn.T, Vm.T* ; Vector half precision

FRSQRTS *Vd.T, Vn.T, Vm.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register

*Vm*

Is the name of the second SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD and FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD and 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* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.99 FSQRT (vector)

Floating-point Square Root (vector).

### Syntax

```
FSQRT Vd.T, Vn.T ; Half-precision  
FSQRT Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.100 FSUB (vector)

Floating-point Subtract (vector).

### Syntax

FSUB *Vd.T, Vn.T, Vm.T* ; Half-precision

FSUB *Vd.T, Vn.T, Vm.T* ; Single-precision and double-precision

Where:

*T*

Is an arrangement specifier:

#### Half-precision

Can be one of 4H or 8H.

#### Single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vd*

Is the name of the SIMD and FP destination register.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD and FP register, from the corresponding elements in the vector in the first source SIMD and FP register, places each result into elements of a vector, and writes the vector to the destination SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.101 INS (vector, element)

Insert vector element from another vector element.

This instruction is used by the alias `MOV` (element).

### Syntax

`INS Vd.Ts[index1], Vn.Ts[index2]`

Where:

`Vd`

Is the name of the SIMD and FP destination register.

`Ts`

Is an element size specifier, and can be one of the values shown in Usage.

`index1`

Is the destination element index, in the range shown in Usage.

`Vn`

Is the name of the SIMD and FP source register.

`index2`

Is the source element index in the range shown in Usage.

### Usage

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD and FP register to the specified vector element of the destination SIMD and FP register.

This instruction can insert data into individual elements within a SIMD and FP register without clearing the remaining bits to zero.

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.

The following table shows the valid specifier combinations:

**Table D6-16 INS (Vector) specifier combinations**

<code>Ts</code>	<code>index1</code>	<code>index2</code>
B	0 to 15	0 to 15
H	0 to 7	0 to 7
S	0 to 3	0 to 3
D	0 or 1	0 or 1

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.102 INS (vector, general)

Insert vector element from general-purpose register.

This instruction is used by the alias MOV (from general).

### Syntax

INS *Vd.Ts[index], Rn*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ts*

Is an element size specifier, and can be one of the values shown in Usage.

*index*

Is the element index, in the range shown in Usage.

*R*

Is the width specifier for the general-purpose source register, and can be either W or X.

*n*

Is the number [0-30] of the general-purpose source register or ZR (31).

### Usage

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD and FP register.

This instruction can insert data into individual elements within a SIMD and FP register without clearing the remaining bits to zero.

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.

The following table shows the valid specifier combinations:

Table D6-17 INS (Vector) specifier combinations

<i>Ts</i>	<i>index</i>	<i>R</i>
B	0 to 15	W
H	0 to 7	W
S	0 to 3	W
D	0 or 1	X

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.103 LD1 (vector, multiple structures)

Load multiple single-element structures to one, two, three, or four registers.

### Syntax

```

LD1 { Vt.T }, [Xn/SP] ; One register
LD1 { Vt.T, Vt2.T }, [Xn/SP] ; Two registers
LD1 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP] ; Three registers
LD1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP] ; Four registers
LD1 { Vt.T }, [Xn/SP], imm ; One register, immediate offset, Post-index
LD1 { Vt.T }, [Xn/SP], Xm ; One register, register offset, Post-index
LD1 { Vt.T, Vt2.T }, [Xn/SP], imm ; Two registers, immediate offset, Post-index
LD1 { Vt.T, Vt2.T }, [Xn/SP], Xm ; Two registers, register offset, Post-index
LD1 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], imm ; Three registers, immediate offset, Post-index
LD1 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], Xm ; Three registers, register offset, Post-index
LD1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], imm ; Four registers, immediate offset, Post-index
LD1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], Xm ; Four registers, register offset, Post-index

```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**Vt4**

Is the name of the fourth SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset:

**One register, immediate offset**

Can be one of #8 or #16.

**Two registers, immediate offset**

Can be one of #16 or #32.

**Three registers, immediate offset**

Can be one of #24 or #48.

**Four registers, immediate offset**

Can be one of #32 or #64.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

## Usage

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 and 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.

The following tables show valid specifier combinations:

**Table D6-18 LD1 (One register, immediate offset) specifier combinations**

<i>T</i>	<i>imm</i>
8B	#8
16B	#16
4H	#8
8H	#16
2S	#8
4S	#16
1D	#8
2D	#16

**Table D6-19 LD1 (Two registers, immediate offset) specifier combinations**

<i>T</i>	<i>imm</i>
8B	#16
16B	#32
4H	#16
8H	#32
2S	#16
4S	#32
1D	#16
2D	#32

**Table D6-20 LD1 (Three registers, immediate offset) specifier combinations**

<i>T</i>	<i>imm</i>
8B	#24
16B	#48
4H	#24
8H	#48
2S	#24
4S	#48

**Table D6-20 LD1 (Three registers, immediate offset) specifier combinations (continued)**

<i>T</i>	<i>imm</i>
1D	#24
2D	#48

**Table D6-21 LD1 (Four registers, immediate offset) specifier combinations**

<i>T</i>	<i>imm</i>
8B	#32
16B	#64
4H	#32
8H	#64
2S	#32
4S	#64
1D	#32
2D	#64

***Related reference***

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.104 LD1 (vector, single structure)

Load one single-element structure to one lane of one register.

### Syntax

```
LD1 { Vt.B }[index], [Xn/SP] ; 8-bit  
LD1 { Vt.H }[index], [Xn/SP] ; 16-bit  
LD1 { Vt.S }[index], [Xn/SP] ; 32-bit  
LD1 { Vt.D }[index], [Xn/SP] ; 64-bit  
LD1 { Vt.B }[index], [Xn/SP], #1 ; 8-bit, immediate offset, Post-index  
LD1 { Vt.B }[index], [Xn/SP], Xm ; 8-bit, register offset, Post-index  
LD1 { Vt.H }[index], [Xn/SP], #2 ; 16-bit, immediate offset, Post-index  
LD1 { Vt.H }[index], [Xn/SP], Xm ; 16-bit, register offset, Post-index  
LD1 { Vt.S }[index], [Xn/SP], #4 ; 32-bit, immediate offset  
LD1 { Vt.S }[index], [Xn/SP], Xm ; 32-bit, register offset  
LD1 { Vt.D }[index], [Xn/SP], #8 ; 64-bit, immediate offset  
LD1 { Vt.D }[index], [Xn/SP], Xm ; 64-bit, register offset
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**index**

The value depends on the instruction variant:

**8-bit**

Is the element index, in the range 0 to 15.

**16-bit**

Is the element index, in the range 0 to 7.

**32-bit**

Is the element index, in the range 0 to 3.

**64-bit**

Is the element index, and can be either 0 or 1.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

### Usage

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 and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.105 LD1R (vector)

Load one single-element structure and Replicate to all lanes (of one register).

### Syntax

```
LD1R { Vt.T }, [Xn/SP] ; No offset
LD1R { Vt.T }, [Xn/SP], imm ; Immediate offset, Post-index
LD1R { Vt.T }, [Xn/SP], Xm ; Register offset, Post-index
```

Where:

*imm*

Is the post-index immediate offset, and can be one of the values shown in Usage.

*Xm*

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

*Vt*

Is the name of the first or only SIMD and FP register to be transferred.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Xn/SP*

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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 and 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.

The following table shows the valid specifier combinations:

Table D6-22 LD1R (Immediate offset) specifier combinations

<i>T</i>	<i>imm</i>
8B	#1
16B	#1
4H	#2
8H	#2
2S	#4
4S	#4
1D	#8
2D	#8

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.106 LD2 (vector, multiple structures)

Load multiple 2-element structures to two registers.

### Syntax

```
LD2 { Vt.T, Vt2.T }, [Xn/SP] ; No offset  
LD2 { Vt.T, Vt2.T }, [Xn/SP], imm ; Immediate offset, Post-index  
LD2 { Vt.T, Vt2.T }, [Xn/SP], Xm ; Register offset, Post-index
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be either #16 or #32.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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 and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.107 LD2 (vector, single structure)

Load single 2-element structure to one lane of two registers.

### Syntax

```
LD2 { Vt.B, Vt2.B }[index], [Xn/SP] ; 8-bit
LD2 { Vt.H, Vt2.H }[index], [Xn/SP] ; 16-bit
LD2 { Vt.S, Vt2.S }[index], [Xn/SP] ; 32-bit
LD2 { Vt.D, Vt2.D }[index], [Xn/SP] ; 64-bit
LD2 { Vt.B, Vt2.B }[index], [Xn/SP], #2 ; 8-bit, immediate offset, Post-index
LD2 { Vt.B, Vt2.B }[index], [Xn/SP], Xm ; 8-bit, register offset, Post-index
LD2 { Vt.H, Vt2.H }[index], [Xn/SP], #4 ; 16-bit, immediate offset, Post-index
LD2 { Vt.H, Vt2.H }[index], [Xn/SP], Xm ; 16-bit, register offset, Post-index
LD2 { Vt.S, Vt2.S }[index], [Xn/SP], #8 ; 32-bit, immediate offset
LD2 { Vt.S, Vt2.S }[index], [Xn/SP], Xm ; 32-bit, register offset
LD2 { Vt.D, Vt2.D }[index], [Xn/SP], #16 ; 64-bit, immediate offset
LD2 { Vt.D, Vt2.D }[index], [Xn/SP], Xm ; 64-bit, register offset
```

Where:

#### Vt

Is the name of the first or only SIMD and FP register to be transferred.

#### Vt2

Is the name of the second SIMD and FP register to be transferred.

#### index

The value depends on the instruction variant:

##### 8-bit

Is the element index, in the range 0 to 15.

##### 16-bit

Is the element index, in the range 0 to 7.

##### 32-bit

Is the element index, in the range 0 to 3.

##### 64-bit

Is the element index, and can be either 0 or 1.

#### Xn/SP

Is the 64-bit name of the general-purpose base register or stack pointer.

#### Xm

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

### Usage

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 and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.108 LD2R (vector)

Load single 2-element structure and Replicate to all lanes of two registers.

### Syntax

```
LD2R { Vt.T, Vt2.T }, [Xn/SP] ; No offset
LD2R { Vt.T, Vt2.T }, [Xn/SP], imm ; Immediate offset, Post-index
LD2R { Vt.T, Vt2.T }, [Xn/SP], Xm ; Register offset, Post-index
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be one of the values shown in Usage.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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 and 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.

The following table shows the valid specifier combinations:

**Table D6-23 LD2R (Immediate offset) specifier combinations**

<b>T</b>	<b>imm</b>
8B	#2
16B	#2
4H	#4
8H	#4
2S	#8
4S	#8
1D	#16
2D	#16

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.109 LD3 (vector, multiple structures)

Load multiple 3-element structures to three registers.

### Syntax

```
LD3 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP] ; No offset  
LD3 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], imm ; Immediate offset, Post-index  
LD3 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], Xm ; Register offset, Post-index
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be either #24 or #48.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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 and FP registers, with de-interleaving.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.110 LD3 (vector, single structure)

Load single 3-element structure to one lane of three registers).

### Syntax

```
LD3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn/SP] ; 8-bit
LD3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn/SP] ; 16-bit
LD3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn/SP] ; 32-bit
LD3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn/SP] ; 64-bit
LD3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn/SP], #3 ; 8-bit, immediate offset, Post-index
LD3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn/SP], Xm ; 8-bit, register offset, Post-index
LD3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn/SP], #6 ; 16-bit, immediate offset, Post-index
LD3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn/SP], Xm ; 16-bit, register offset, Post-index
LD3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn/SP], #12 ; 32-bit, immediate offset
LD3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn/SP], Xm ; 32-bit, register offset
LD3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn/SP], #24 ; 64-bit, immediate offset
LD3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn/SP], Xm ; 64-bit, register offset
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**index**

The value depends on the instruction variant:

**8-bit**

Is the element index, in the range 0 to 15.

**16-bit**

Is the element index, in the range 0 to 7.

**32-bit**

Is the element index, in the range 0 to 3.

**64-bit**

Is the element index, and can be either 0 or 1.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

### Usage

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 and 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.

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.111 LD3R (vector)

Load single 3-element structure and Replicate to all lanes of three registers.

### Syntax

```
LD3R { Vt.T, Vt2.T, Vt3.T }, [Xn/SP] ; No offset
LD3R { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], imm ; Immediate offset, Post-index
LD3R { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], Xm ; Register offset, Post-index
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be one of the values shown in Usage.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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 and 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.

The following table shows the valid specifier combinations:

Table D6-24 LD3R (Immediate offset) specifier combinations

T	imm
8B	#3
16B	#3
4H	#6
8H	#6
2S	#12
4S	#12
1D	#24
2D	#24

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.112 LD4 (vector, multiple structures)

Load multiple 4-element structures to four registers.

### Syntax

```
LD4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP] ; No offset  
LD4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], imm ; Immediate offset, Post-index  
LD4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], Xm ; Register offset, Post-index
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**Vt4**

Is the name of the fourth SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be either #32 or #64.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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 and FP registers, with de-interleaving.

For an example of de-interleaving, see [LD3 \(multiple structures\)](#).

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.113 LD4 (vector, single structure)

Load single 4-element structure to one lane of four registers.

### Syntax

```

LD4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn/SP] ; 8-bit
LD4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn/SP] ; 16-bit
LD4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn/SP] ; 32-bit
LD4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn/SP] ; 64-bit
LD4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn/SP], #4 ; 8-bit, immediate offset,
Post-index
LD4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn/SP], Xm ; 8-bit, register offset, Post-
index
LD4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn/SP], #8 ; 16-bit, immediate offset,
Post-index
LD4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn/SP], Xm ; 16-bit, register offset,
Post-index
LD4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn/SP], #16 ; 32-bit, immediate offset
LD4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn/SP], Xm ; 32-bit, register offset
LD4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn/SP], #32 ; 64-bit, immediate offset
LD4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn/SP], Xm ; 64-bit, register offset

```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**Vt4**

Is the name of the fourth SIMD and FP register to be transferred.

**index**

The value depends on the instruction variant:

**8-bit**

Is the element index, in the range 0 to 15.

**16-bit**

Is the element index, in the range 0 to 7.

**32-bit**

Is the element index, in the range 0 to 3.

**64-bit**

Is the element index, and can be either 0 or 1.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

## Usage

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 and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.114 LD4R (vector)

Load single 4-element structure and Replicate to all lanes of four registers.

### Syntax

```
LD4R { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP] ; No offset
LD4R { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], imm ; Immediate offset, Post-index
LD4R { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], Xm ; Register offset, Post-index
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**Vt4**

Is the name of the fourth SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be one of the values shown in Usage.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

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 and 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.

The following table shows the valid specifier combinations:

Table D6-25 LD4R (Immediate offset) specifier combinations

T	imm
8B	#4
16B	#4
4H	#8
8H	#8
2S	#16
4S	#16
1D	#32
2D	#32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.115 MLA (vector, by element)

Multiply-Add to accumulator (vector, by element).

### Syntax

`MLA Vd.T, Vn.T, Vm.Ts[index]`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If **Ts** is H, then **Vm** must be in the range V0 to V15.
- If **Ts** is S, then **Vm** must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the results with the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-26 MLA (Vector) specifier combinations**

<b>T</b>	<b>Ts</b>	<b>index</b>
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.116 MLA (vector)

Multiply-Add to accumulator (vector).

### Syntax

MLA  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, and accumulates the results with the vector elements of the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.117 MLS (vector, by element)

Multiply-Subtract from accumulator (vector, by element).

### Syntax

MLS  $Vd.T, Vn.T, Vm.Ts[index]$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of the values shown in Usage.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register:

- If  $Ts$  is H, then  $Vm$  must be in the range V0 to V15.
- If  $Ts$  is S, then  $Vm$  must be in the range V0 to V31.

$Ts$

Is an element size specifier, and can be either H or S.

$index$

Is the element index, in the range shown in Usage.

### Usage

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and subtracts the results from the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

Table D6-27 MLS (Vector) specifier combinations

$T$	$Ts$	$index$
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.118 MLS (vector)

Multiply-Subtract from accumulator (vector).

### Syntax

MLS  $Vd.T, Vn.T,Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, and subtracts the results from the vector elements of the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.119 MOV (vector, element)

Move vector element to another vector element.

This instruction is an alias of **INS** (element).

The equivalent instruction is **INS Vd.Ts[index1], Vn.Ts[index2]**.

### Syntax

**MOV Vd.Ts[index1], Vn.Ts[index2]**

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**Ts**

Is an element size specifier, and can be one of the values shown in Usage.

**index1**

Is the destination element index, in the range shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**index2**

Is the source element index in the range shown in Usage.

### Usage

Move vector element to another vector element. This instruction copies the vector element of the source SIMD and FP register to the specified vector element of the destination SIMD and FP register.

This instruction can insert data into individual elements within a SIMD and FP register without clearing the remaining bits to zero.

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.

The following table shows the valid specifier combinations:

**Table D6-28 MOV (Vector) specifier combinations**

Ts	index1	index2
B	0 to 15	0 to 15
H	0 to 7	0 to 7
S	0 to 3	0 to 3
D	0 or 1	0 or 1

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.120 MOV (vector, from general)

Move general-purpose register to a vector element.

This instruction is an alias of **INS** (general).

The equivalent instruction is **INS Vd.Ts[index], Rn**.

### Syntax

**MOV Vd.Ts[index], Rn**

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**Ts**

Is an element size specifier, and can be one of the values shown in Usage.

**index**

Is the element index, in the range shown in Usage.

**R**

Is the width specifier for the general-purpose source register, and can be either W or X.

**n**

Is the number [0-30] of the general-purpose source register or ZR (31).

### Usage

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD and FP register.

This instruction can insert data into individual elements within a SIMD and FP register without clearing the remaining bits to zero.

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.

The following table shows the valid specifier combinations:

**Table D6-29 MOV (Vector) specifier combinations**

Ts	index	R
B	0 to 15	W
H	0 to 7	W
S	0 to 3	W
D	0 or 1	X

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.121 MOV (vector)

Move vector.

This instruction is an alias of ORR (vector, register).

The equivalent instruction is ORR  $Vd.T$ ,  $Vn.T$ ,  $Vn.T$ .

### Syntax

`MOV Vd.T, Vn.T`

Where:

**$Vd$**

Is the name of the SIMD and FP destination register.

**$T$**

Is an arrangement specifier, and can be either 8B or 16B.

**$Vn$**

Is the name of the first SIMD and FP source register.

### Usage

Move vector. This instruction copies the vector in the source SIMD and FP register into the destination SIMD and 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.

#### *Related reference*

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.122 MOV (vector, to general)

Move vector element to general-purpose register.

This instruction is an alias of UMOV.

The equivalent instruction is UMOV *Wd*, *Vn.S[index]*.

### Syntax

MOV *Wd*, *Vn.S[index]* ; 32-bit

MOV *Xd*, *Vn.D[index]* ; 64-bit

Where:

*Wd*

Is the 32-bit name of the general-purpose destination register.

*index*

The value depends on the instruction variant:

**32-bit**

Is the element index.

**64-bit**

Is the element index and can be either 0 or 1.

*Xd*

Is the 64-bit name of the general-purpose destination register.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD and FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.123 MOVI (vector)

Move Immediate (vector).

### Syntax

```
MOVI Vd.T, #imm8{, LSL #0} ; 8-bit
MOVI Vd.T, #imm8{, LSL #amount} ; 16-bit shifted immediate
MOVI Vd.T, #imm8{, LSL #amount} ; 32-bit shifted immediate
MOVI Vd.T, #imm8, MSL #amount ; 32-bit shifting ones
MOVI Dd, #imm ; 64-bit scalar
MOVI Vd.2D, #imm ; 64-bit vector
```

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

**8-bit**

Can be one of 8B or 16B.

**16-bit shifted immediate**

Can be one of 4H or 8H.

**32-bit shifted immediate**

Can be one of 2S or 4S.

**32-bit shifting ones**

Can be one of 2S or 4S.

*imm8*

Is an 8-bit immediate.

*amount*

Is the shift amount:

**16-bit shifted immediate**

Can be one of 0 or 8.

**32-bit shifted immediate**

Can be one of 0, 8, 16 or 24.

**32-bit shifting ones**

Can be one of 8 or 16.

Defaults to zero if LSL is omitted.

*Dd*

Is the 64-bit name of the SIMD and FP destination register.

*imm*

Is a 64-bit immediate.

### Usage

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD and 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.

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.124 MUL (vector, by element)

Multiply (vector, by element).

### Syntax

**MUL Vd.T, Vn.T, Vm.Ts[index]**

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If **Ts** is H, then **Vm** must be in the range V0 to V15.
- If **Ts** is S, then **Vm** must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-30 MUL (Vector) specifier combinations**

<b>T</b>	<b>Ts</b>	<b>index</b>
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.125 MUL (vector)

Multiply (vector).

### Syntax

`MUL Vd.T, Vn.T, Vm.T`

Where:

`Vd`

Is the name of the SIMD and FP destination register.

`T`

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

`Vn`

Is the name of the first SIMD and FP source register.

`Vm`

Is the name of the second SIMD and FP source register.

### Usage

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, places the results in a vector, and writes the vector to the destination SIMD and FP register.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.126 MVN (vector)

Bitwise NOT (vector).

This instruction is an alias of **NOT**.

The equivalent instruction is **NOT Vd.T, Vn.T**.

### Syntax

**MVN Vd.T, Vn.T**

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be either 8B or 16B.

**Vn**

Is the name of the SIMD and FP source register.

### Usage

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD and FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.127 MVNI (vector)

Move inverted Immediate (vector).

### Syntax

```
MVNI Vd.T, #imm8{, LSL #amount} ; 16-bit shifted immediate  
MVNI Vd.T, #imm8{, LSL #amount} ; 32-bit shifted immediate  
MVNI Vd.T, #imm8, MSL #amount ; 32-bit shifting ones
```

Where:

*T*

Is an arrangement specifier:

#### 16-bit shifted immediate

Can be one of 4H or 8H.

#### 32-bit shifted immediate

Can be one of 2S or 4S.

#### 32-bit shifting ones

Can be one of 2S or 4S.

*amount*

Is the shift amount:

#### 16-bit shifted immediate

Can be one of 0 or 8.

#### 32-bit shifted immediate

Can be one of 0, 8, 16 or 24.

#### 32-bit shifting ones

Can be one of 8 or 16.

Defaults to zero if LSL is omitted.

*Vd*

Is the name of the SIMD and FP destination register.

*imm8*

Is an 8-bit immediate.

### Usage

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.128 NEG (vector)

Negate (vector).

### Syntax

NEG  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Negate (vector). This instruction reads each vector element from the source SIMD and FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.129 NOT (vector)

Bitwise NOT (vector).

This instruction is used by the alias MVN.

### Syntax

NOT  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be either 8B or 16B.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD and FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.130 ORN (vector)

Bitwise inclusive OR NOT (vector).

### Syntax

ORN  $Vd.T, Vn.T,Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be either 8B or 16B.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD and FP registers, and writes the result to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.131 ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate).

### Syntax

ORR *Vd.T, #imm8{, LSL #amount} ; 16-bit*

ORR *Vd.T, #imm8{, LSL #amount} ; 32-bit*

Where:

*T*

Is an arrangement specifier:

#### 16-bit

Can be one of 4H or 8H.

#### 32-bit

Can be one of 2S or 4S.

*amount*

Is the shift amount:

#### 16-bit

Can be one of 0 or 8.

#### 32-bit

Can be one of 0, 8, 16 or 24.

Defaults to zero if LSL is omitted.

*Vd*

Is the name of the SIMD and FP register.

*imm8*

Is an 8-bit immediate.

### Usage

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD and 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 and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.132 ORR (vector, register)

Bitwise inclusive OR (vector, register).

This instruction is used by the alias `MOV` (vector).

### Syntax

`ORR Vd.T, Vn.T,Vm.T`

Where:

`Vd`

Is the name of the SIMD and FP destination register.

`T`

Is an arrangement specifier, and can be either `8B` or `16B`.

`Vn`

Is the name of the first SIMD and FP source register.

`Vm`

Is the name of the second SIMD and FP source register.

### Usage

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.133 PMUL (vector)

Polynomial Multiply.

### Syntax

PMUL *Vd.T*, *Vn.T*, *Vm.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be either 8B or 16B.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Usage

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, places the results in a vector, and writes the vector to the destination SIMD and FP register.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.134 PMULL, PMULL2 (vector)

Polynomial Multiply Long.

### Syntax

PMULL{2} *Vd.Ta, Vn.Tb, Vm.Tb*

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, 8H.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD and FP registers, places the results in a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

The PMULL instruction extracts each source vector from the lower half of each source register, while the PMULL2 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.

The following table shows the valid specifier combinations:

Table D6-31 PMULL, PMULL2 (Vector) specifier combinations

<Q>	Ta	Tb
-	8H	8B
2	8H	16B

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.135 RADDHN, RADDHN2 (vector)

Rounding Add returning High Narrow.

### Syntax

`RADDHN{2} Vd.Tb, Vn.Ta, Vm.Ta`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD and FP register to the corresponding vector element in the second source SIMD and FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register.

The results are rounded. For truncated results, see [D6.4 ADDHN, ADDHN2 \(vector\) on page D6-1256](#).

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 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.

The following table shows the valid specifier combinations:

**Table D6-32 RADDHN, RADDHN2 (Vector) specifier combinations**

<Q>	Tb	Ta
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.136 RBIT (vector)

Reverse Bit order (vector).

### Syntax

RBIT *Vd.T, Vn.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be either 8B or 16B.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD and FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD and FP register.

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

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.137 REV16 (vector)

Reverse elements in 16-bit halfwords (vector).

### Syntax

REV16 *Vd.T, Vn.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be either 8B or 16B.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

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 and FP register, places the results into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.138 REV32 (vector)

Reverse elements in 32-bit words (vector).

### Syntax

REV32 *Vd.T, Vn.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H or 8H.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

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 and FP register, places the results into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.139 REV64 (vector)

Reverse elements in 64-bit doublewords (vector).

### Syntax

REV64 *Vd.T, Vn.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

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 and FP register, places the results into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.140 RSHRN, RSHRN2 (vector)

Rounding Shift Right Narrow (immediate).

### Syntax

`RSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

### Usage

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see [D6.158 SHRN, SHRN2 \(vector\) on page D6-1420](#).

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 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.

The following table shows the valid specifier combinations:

**Table D6-33 RSHRN, RSHRN2 (Vector) specifier combinations**

<Q>	Tb	Ta	shift
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.141 RSUBHN, RSUBHN2 (vector)

Rounding Subtract returning High Narrow.

### Syntax

`RSUBHN{2} Vd.Tb, Vn.Ta, Vm.Ta`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD and FP register from the corresponding vector element of the first source SIMD and FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register.

The results are rounded. For truncated results, see [D6.222 SUBHN, SUBHN2 \(vector\) on page D6-1491](#).

The RSUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSUBHN2 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.

The following table shows the valid specifier combinations:

**Table D6-34 RSUBHN, RSUBHN2 (Vector) specifier combinations**

<Q>	Tb	Ta
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.142 SABA (vector)

Signed Absolute difference and Accumulate.

### Syntax

SABA  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD and FP register from the corresponding elements of the first source SIMD and FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.143 SABAL, SABAL2 (vector)

Signed Absolute difference and Accumulate Long.

### Syntax

`SABAL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See `<Q>` in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD and FP register from the corresponding vector elements of the first source SIMD and FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-35 SABAL, SABAL2 (Vector) specifier combinations**

<code>&lt;Q&gt;</code>	<code>Ta</code>	<code>Tb</code>
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.144 SABD (vector)

Signed Absolute Difference.

### Syntax

SABD  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD and FP register from the corresponding elements of the first source SIMD and FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.145 SABDL, SABDL2 (vector)

Signed Absolute Difference Long.

### Syntax

`SABDL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD and FP register from the corresponding vector elements of the first source SIMD and 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 and 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.

The following table shows the valid specifier combinations:

**Table D6-36 SABDL, SABDL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.146 SADALP (vector)

Signed Add and Accumulate Long Pairwise.

### Syntax

SADALP *Vd.Ta, Vn.Tb*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD and FP register and accumulates the results into the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-37 SADALP (Vector) specifier combinations**

<i>Ta</i>	<i>Tb</i>
4H	8B
8H	16B
2S	4H
4S	8H
1D	2S
2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.147 SADDL, SADDL2 (vector)

Signed Add Long (vector).

### Syntax

SADDL{2}  $Vd.Ta, Vn.Tb, Vm.Tb$

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD and FP register to the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register, while the SADDL2 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.

The following table shows the valid specifier combinations:

**Table D6-38 SADDL, SADDL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.148 SADDLP (vector)

Signed Add Long Pairwise.

### Syntax

SADDLP *Vd.Ta, Vn.Tb*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-39 SADDLP (Vector) specifier combinations**

<i>Ta</i>	<i>Tb</i>
4H	8B
8H	16B
2S	4H
4S	8H
1D	2S
2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.149 SADDLV (vector)

Signed Add Long across Vector.

### Syntax

SADDLV *Vd, Vn.T*

Where:

*V*

Is the destination width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD and FP register together, and writes the scalar result to the destination SIMD and FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed 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.

The following table shows the valid specifier combinations:

**Table D6-40 SADDLV (Vector) specifier combinations**

<i>V</i>	<i>T</i>
H	8B
H	16B
S	4H
S	8H
D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.150 SADDW, SADDW2 (vector)

Signed Add Wide.

### Syntax

`SADDW{2} Vd.Ta, Vn.Ta, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed Add Wide. This instruction adds vector elements of the first source SIMD and FP register to the corresponding vector elements in the lower or upper half of the second source SIMD and FP register, places the results in a vector, and writes the vector to the SIMD and FP destination register.

The SADDW instruction extracts the second source vector from the lower half of the second source register, while the SADDW2 instruction extracts the second source vector from the upper half of the second 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.

The following table shows the valid specifier combinations:

**Table D6-41 SADDW, SADDW2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.151 SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector).

### Syntax

SCVTF *Vd.T, Vn.T, #fbits*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*fbits*

Is the number of fractional bits, in the range 1 to the element width.

### Usage

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-42 SCVTF (Vector) specifier combinations**

<i>T</i>	<i>fbits</i>
4H	
8H	
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.152 SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector).

### Syntax

SCVTF *Vd.T, Vn.T* ; Vector half precision

SCVTF *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.153 SDOT (vector, by element)

Dot Product signed arithmetic (vector, by element).

### Syntax

SDOT *Vd.Ta, Vn.Tb, Vm.4B[index]*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be either 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be either 8B or 16B.

*Vm*

Is the name of the second SIMD and FP source register in the range 0 to 31.

*index*

Is the element index.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

### Usage

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the 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.

---

#### Note

---

ID\_AA64ISAR0\_EL1.DP indicates whether this instruction is supported. See *ID\_AA64ISAR0\_EL1* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.154 SDOT (vector)

Dot Product signed arithmetic (vector).

### Syntax

SDOT *Vd.Ta, Vn.Tb, Vm.Tb*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be either 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be either 8B or 16B.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

For Armv8.2 and Armv8.2, this is an OPTIONAL instruction.

### Usage

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the 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.

---

#### Note

---

ID\_AA64ISAR0\_EL1.DP indicates whether this instruction is supported. See *ID\_AA64ISAR0\_EL1* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

#### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.155 SHADD (vector)

Signed Halving Add.

### Syntax

SHADD  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD and FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see [D6.202 SRHADD \(vector\) on page D6-1467](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.156 SHL (vector)

Shift Left (immediate).

### Syntax

SHL *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Shift Left (immediate). This instruction reads each value from a vector, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

Table D6-43 SHL (Vector) specifier combinations

<i>T</i>	<i>shift</i>
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.157 SHLL, SHLL2 (vector)

Shift Left Long (by element size).

### Syntax

`SHLL{2} Vd.Ta, Vn.Tb, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the left shift amount, which must be equal to the source element width in bits, and can be one of the values shown in Usage.

### Usage

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD and FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

The SHLL instruction extracts vector elements from the lower half of the source register, while the SHLL2 instruction extracts vector elements from the upper half of the 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.

The following table shows the valid specifier combinations:

**Table D6-44 SHLL, SHLL2 (Vector) specifier combinations**

<Q>	Ta	Tb	shift
-	8H	8B	8
2	8H	16B	8
-	4S	4H	16
2	4S	8H	16
-	2D	2S	32
2	2D	4S	32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.158 SHRN, SHRN2 (vector)

Shift Right Narrow (immediate).

### Syntax

`SHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

### Usage

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD and FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see [D6.140 RSHRN, RSHRN2 \(vector\) on page D6-1402](#).

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 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.

The following table shows the valid specifier combinations:

**Table D6-45 SHRN, SHRN2 (Vector) specifier combinations**

<Q>	Tb	Ta	shift
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.159 SHSUB (vector)

Signed Halving Subtract.

### Syntax

SHSUB  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD and FP register from the corresponding elements in the vector in the first source SIMD and FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD and 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.

### *Related reference*

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.160 SLI (vector)

Shift Left and Insert (immediate).

### Syntax

`SLI Vd.T, Vn.T, #shift`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**shift**

Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD and FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD and FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

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.

The following table shows the valid specifier combinations:

**Table D6-46 SLI (Vector) specifier combinations**

<b>T</b>	<b>shift</b>
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.161 SMAX (vector)

Signed Maximum (vector).

### Syntax

SMAX  $Vd.T$ ,  $Vn.T$ ,  $Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD and FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.162 SMAXP (vector)

Signed Maximum Pairwise.

### Syntax

SMAXP  $Vd.T$ ,  $Vn.T$ ,  $Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.163 SMAXV (vector)

Signed Maximum across Vector.

### Syntax

SMAXV *Vd, Vn.T*

Where:

*V*

Is the destination width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the largest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are signed integer values.

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

The following table shows the valid specifier combinations:

Table D6-47 SMAXV (Vector) specifier combinations

<i>V</i>	<i>T</i>
B	8B
B	16B
H	4H
H	8H
S	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.164 SMIN (vector)

Signed Minimum (vector).

### Syntax

SMIN  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD and FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD and FP register.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.165 SMINP (vector)

Signed Minimum Pairwise.

### Syntax

`SMINP Vd.T, Vn.T, Vm.T`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.166 SMINV (vector)

Signed Minimum across Vector.

### Syntax

`SMINV Vd, Vn.T`

Where:

**V**

Is the destination width specifier, and can be one of the values shown in Usage.

**d**

Is the number of the SIMD and FP destination register.

**Vn**

Is the name of the SIMD and FP source register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the smallest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are signed integer values.

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

The following table shows the valid specifier combinations:

**Table D6-48 SMINV (Vector) specifier combinations**

<b>V</b>	<b>T</b>
B	8B
B	16B
H	4H
H	8H
S	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.167 SMLAL, SMLAL2 (vector, by element)

Signed Multiply-Add Long (vector, by element).

### Syntax

`SMLAL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element in the second source SIMD and FP register, and accumulates the results with the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-49 SMLAL, SMLAL2 (Vector) specifier combinations**

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.168 SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector).

### Syntax

SMLAL{2} *Vd.Ta, Vn.Tb, Vm.Tb*

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

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 and FP registers, and accumulates the results with the vector elements of the destination SIMD and 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\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table D6-50 SMLAL, SMLAL2 (Vector) specifier combinations

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.169 SMLSL, SMLSL2 (vector, by element)

Signed Multiply-Subtract Long (vector, by element).

### Syntax

`SMLSL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register and subtracts the results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts vector elements from the lower half of the first source register, while the SMLSL2 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.

The following table shows the valid specifier combinations:

**Table D6-51 SMLSL, SMLSL2 (Vector) specifier combinations**

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.170 SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector).

### Syntax

`SMLSL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

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 and FP registers, and subtracts the results from the vector elements of the destination SIMD and 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\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

**Table D6-52 SMLSL, SMLSL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.171 SMOV (vector)

Signed Move vector element to general-purpose register.

### Syntax

`SMOV Wd, Vn.Ts[index] ; 32-bit`

`SMOV Xd, Vn.Ts[index] ; 64-bit`

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Ts**

Is an element size specifier:

**32-bit**

Can be one of B or H.

**64-bit**

Can be one of B, H or S.

**index**

Is the element index, in the range shown in Usage.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Usage

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD and FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose 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.

The following tables show valid specifier combinations:

**Table D6-53 SMOV (32-bit) specifier combinations**

Ts	index
B	0 to 15
H	0 to 7

**Table D6-54 SMOV (64-bit) specifier combinations**

Ts	index
B	0 to 15
H	0 to 7
S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.172 SMULL, SMULL2 (vector, by element)

Signed Multiply Long (vector, by element).

### Syntax

`SMULL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

Table D6-55 SMULL, SMULL2 (Vector) specifier combinations

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.173 SMULL, SMULL2 (vector)

Signed Multiply Long (vector).

### Syntax

SMULL{2} *Vd.Ta, Vn.Tb, Vm.Tb*

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

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 and FP registers, places the results in a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-56 SMULL, SMULL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.174 SQABS (vector)

Signed saturating Absolute value.

### Syntax

SQABS  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD and FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.175 SQADD (vector)

Signed saturating Add.

### Syntax

SQADD  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.176 SQDMLAL, SQDMLAL2 (vector, by element)

Signed saturating Doubling Multiply-Add Long (by element).

### Syntax

`SQDMLAL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

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 and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-57 SQDMLAL{2} (Vector) specifier combinations**

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.177 SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long.

### Syntax

`SQDMLAL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

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 and FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

Table D6-58 SQDMLAL{2} (Vector) specifier combinations

<Q>	Ta	Tb
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.178 SQDMLSL, SQDMLSL2 (vector, by element)

Signed saturating Doubling Multiply-Subtract Long (by element).

### Syntax

`SQDMLSL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

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 and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-59 SQDMLSL{2} (Vector) specifier combinations**

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.179 SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long.

### Syntax

`SQDMLSL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

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 and FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-60 SQDMLSL{2} (Vector) specifier combinations**

<Q>	Ta	Tb
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.180 SQDMULH (vector, by element)

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

### Syntax

SQDMULH  $Vd.T, Vn.T, Vm.Ts[index]$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of the values shown in Usage.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register:

- If  $Ts$  is H, then  $Vm$  must be in the range V0 to V15.
- If  $Ts$  is S, then  $Vm$  must be in the range V0 to V31.

$Ts$

Is an element size specifier, and can be either H or S.

$index$

Is the element index, in the range shown in Usage.

### Usage

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see [D6.189 SQRDMULH \(vector, by element\) on page D6-1454](#).

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.

The following table shows the valid specifier combinations:

**Table D6-61 SQDMULH (Vector) specifier combinations**

$T$	$Ts$	$index$
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.181 SQDMULH (vector)

Signed saturating Doubling Multiply returning High half.

### Syntax

SQDMULH  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD and FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see [D6.190 SQRDMULH \(vector\) on page D6-1455](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.182 SQDMULL, SQDMULL2 (vector, by element)

Signed saturating Doubling Multiply Long (by element).

### Syntax

`SQDMULL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD and 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\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

**Table D6-62 SQDMULL{2} (Vector) specifier combinations**

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.183 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long.

### Syntax

`SQDMULL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD and FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-63 SQDMULL{2} (Vector) specifier combinations**

<Q>	Ta	Tb
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.184 SQNEG (vector)

Signed saturating Negate.

### Syntax

SQNEG  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Signed saturating Negate. This instruction reads each vector element from the source SIMD and FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.185 SQRDMLAH (vector, by element)

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

### Syntax

`SQRDMLAH Vd.T, Vn.T, Vm.Ts[index]`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If **Ts** is H, then **Vm** must be in the range V0 to V15.
- If **Ts** is S, then **Vm** must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Architectures supported (vector)

Supported in the Armv8.1 architecture and later.

### Usage

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD and FP register with the value of a vector element of the second source SIMD and FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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.

The following table shows the valid specifier combinations:

**Table D6-64 SQRDMLAH (Vector) specifier combinations**

<b>T</b>	<b>Ts</b>	<b>index</b>
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.186 SQRDMLAH (vector)

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

### Syntax

SQRDMLAH  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.1 architecture and later.

### Usage

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD and FP register with the corresponding vector elements of the second source SIMD and FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.187 SQRDMLSH (vector, by element)

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

### Syntax

`SQRDMLSH Vd.T, Vn.T, Vm.Ts[index]`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If **Ts** is H, then **Vm** must be in the range V0 to V15.
- If **Ts** is S, then **Vm** must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Architectures supported (vector)

Supported in the Armv8.1 architecture and later.

### Usage

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD and FP register with the value of a vector element of the second source SIMD and FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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.

The following table shows the valid specifier combinations:

**Table D6-65 SQRDMLSH (Vector) specifier combinations**

<b>T</b>	<b>Ts</b>	<b>index</b>
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.188 SQRDMLSH (vector)

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

### Syntax

SQRDMLSH  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.1 architecture and later.

### Usage

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD and FP register with the corresponding vector elements of the second source SIMD and FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.189 SQRDMULH (vector, by element)

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

### Syntax

`SQRDMULH Vd.T, Vn.T, Vm.Ts[index]`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register:

- If **Ts** is H, then **Vm** must be in the range V0 to V15.
- If **Ts** is S, then **Vm** must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see [D6.180 SQDMULH \(vector, by element\) on page D6-1444](#).

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D6-66 SQRDMULH (Vector) specifier combinations**

<b>T</b>	<b>Ts</b>	<b>index</b>
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.190 SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half.

### Syntax

`SQRDMULH Vd.T, Vn.T, Vm.T`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of 4H, 8H, 2S or 4S.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD and FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see [D6.181 SQDMULH \(vector\) on page D6-1445](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.191 SQRSHL (vector)

Signed saturating Rounding Shift Left (register).

### Syntax

SQRSHL  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD and FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see [D6.195 SQSHL \(vector, register\) on page D6-1460](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.192 SQRSHRN, SQRSHRN2 (vector)

Signed saturating Rounded Shift Right Narrow (immediate).

### Syntax

`SQRSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

### Usage

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see [D6.197 SQSHRN, SQSHRN2 \(vector\) on page D6-1462](#).

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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

The following table shows the valid specifier combinations:

**Table D6-67 SQRSHRN{2} (Vector) specifier combinations**

<Q>	Tb	Ta	shift
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.193 SQRSHRUN, SQRSHRUN2 (vector)

Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

### Syntax

`SQRSHRUN{2} Vd.Tb, Vn.Ta, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

### Usage

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD and FP register. The results are rounded. For truncated results, see [D6.198 SQSHRUN, SQSHRUN2 \(vector\) on page D6-1463](#).

The SQRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D6-68 SQRSHRUN{2} (Vector) specifier combinations**

<Q>	Tb	Ta	shift
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.194 SQSHL (vector, immediate)

Signed saturating Shift Left (immediate).

### Syntax

SQSHL *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD and FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see [D6.258 UQRSHL \(vector\) on page D6-1527](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

Table D6-69 SQSHL (Vector) specifier combinations

<i>T</i>	<i>shift</i>
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.195 SQSHL (vector, register)

Signed saturating Shift Left (register).

### Syntax

SQSHL  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see [D6.191 SQRSHL \(vector\) on page D6-1456](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.196 SQSHLU (vector)

Signed saturating Shift Left Unsigned (immediate).

### Syntax

SQSHLU *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see [D6.258 UQRSHL \(vector\) on page D6-1527](#).

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

Table D6-70 SQSHLU (Vector) specifier combinations

<i>T</i>	<i>shift</i>
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.197 SQSHRN, SQSHRN2 (vector)

Signed saturating Shift Right Narrow (immediate).

### Syntax

`SQSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

### Usage

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see [D6.192 SQRSHRN, SQRSHRN2 \(vector\) on page D6-1457](#).

The SQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D6-71 SQSHRN{2} (Vector) specifier combinations**

<Q>	Tb	Ta	shift
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.198 SQSHRUN, SQSHRUN2 (vector)

Signed saturating Shift Right Unsigned Narrow (immediate).

### Syntax

`SQSHRUN{2} Vd.Tb, Vn.Ta, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

### Usage

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see [D6.193 SQRSHRUN, SQRSHRUN2 \(vector\) on page D6-1458](#).

The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D6-72 SQSHRUN{2} (Vector) specifier combinations**

<Q>	Tb	Ta	shift
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.199 SQSUB (vector)

Signed saturating Subtract.

### Syntax

SQSUB  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD and FP register from the corresponding element values of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.200 SQXTN, SQXTN2 (vector)

Signed saturating extract Narrow.

### Syntax

$\text{SQXTN}\{2\} \text{ } Vd.Tb, \text{ } Vn.Ta$

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD and FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. 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 SQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTN2 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.

The following table shows the valid specifier combinations:

**Table D6-73 SQXTN{2} (Vector) specifier combinations**

<Q>	Tb	Ta
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.201 SQXTUN, SQXTUN2 (vector)

Signed saturating extract Unsigned Narrow.

### Syntax

`SQXTUN{2} Vd.Tb, Vn.Ta`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD and FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQXTUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTUN2 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.

The following table shows the valid specifier combinations:

**Table D6-74 SQXTUN{2} (Vector) specifier combinations**

<Q>	Tb	Ta
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.202 SRHADD (vector)

Signed Rounding Halving Add.

### Syntax

SRHADD *Vd.T, Vn.T,Vm.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Usage

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD and FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see [D6.155 SHADD \(vector\) on page D6-1417](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.203 SRI (vector)

Shift Right and Insert (immediate).

### Syntax

**SRI *Vd.T, Vn.T, #shift***

Where:

***Vd***

Is the name of the SIMD and FP destination register.

***T***

Is an arrangement specifier, and can be one of the values shown in Usage.

***Vn***

Is the name of the SIMD and FP source register.

***shift***

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD and FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

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.

The following table shows the valid specifier combinations:

**Table D6-75 SRI (Vector) specifier combinations**

<b><i>T</i></b>	<b><i>shift</i></b>
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.204 SRSHL (vector)

Signed Rounding Shift Left (register).

### Syntax

SRSHL  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD and FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see [D6.207 SSHL \(vector\) on page D6-1472](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.205 SRSHR (vector)

Signed Rounding Shift Right (immediate).

### Syntax

SRSHR *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see [D6.209 SSHR \(vector\) on page D6-1474](#).

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.

The following table shows the valid specifier combinations:

**Table D6-76 SRSHR (Vector) specifier combinations**

<i>T</i>	<i>shift</i>
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.206 SRSRA (vector)

Signed Rounding Shift Right and Accumulate (immediate).

### Syntax

SRSRA  $Vd.T, Vn.T, \#shift$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of the values shown in Usage.

$Vn$

Is the name of the SIMD and FP source register.

$shift$

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see [D6.210 SSRA \(vector\) on page D6-1475](#).

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.

The following table shows the valid specifier combinations:

**Table D6-77 SRSRA (Vector) specifier combinations**

$T$	$shift$
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.207 SSHL (vector)

Signed Shift Left (register).

### Syntax

SSHL  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD and FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see [D6.204 SRSHL \(vector\) on page D6-1469](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.208 SSHLL, SSHLL2 (vector)

Signed Shift Left Long (immediate).

This instruction is used by the alias SXTL, SXTL2, SXTL, SXTL22.

### Syntax

`SSHLL{2} Vd.Ta, Vn.Tb, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the left shift amount, in the range 0 to the source element width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD and FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register, while the SSHLL2 instruction extracts vector elements from the upper half of the 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.

The following table shows the valid specifier combinations:

Table D6-78 SSHLL, SSHLL2 (Vector) specifier combinations

<Q>	Ta	Tb	shift
-	8H	8B	0 to 7
2	8H	16B	0 to 7
-	4S	4H	0 to 15
2	4S	8H	0 to 15
-	2D	2S	0 to 31
2	2D	4S	0 to 31

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.209 SSHR (vector)

Signed Shift Right (immediate).

### Syntax

SSHR *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see [D6.205 SRSHR \(vector\) on page D6-1470](#).

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.

The following table shows the valid specifier combinations:

**Table D6-79 SSHR (Vector) specifier combinations**

<i>T</i>	<i>shift</i>
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.210 SSRA (vector)

Signed Shift Right and Accumulate (immediate).

### Syntax

SSRA *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see [D6.206 SRSRA \(vector\) on page D6-1471](#).

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.

The following table shows the valid specifier combinations:

Table D6-80 SSRA (Vector) specifier combinations

<i>T</i>	<i>shift</i>
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.211 SSUBL, SSUBL2 (vector)

Signed Subtract Long.

### Syntax

SSUBL{2}  $Vd.Ta, Vn.Tb, Vm.Tb$

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD and FP register from the corresponding vector element of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register, while the SSUBL2 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.

The following table shows the valid specifier combinations:

Table D6-81 SSUBL, SSUBL2 (Vector) specifier combinations

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.212 SSUBW, SSUBW2 (vector)

Signed Subtract Wide.

### Syntax

`SSUBW{2} Vd.Ta, Vn.Ta, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD and FP register from the corresponding vector element in the first source SIMD and FP register, places the result in a vector, and writes the vector to the SIMD and FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register, while the SSUBW2 instruction extracts the second source vector from the upper half of the second 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.

The following table shows the valid specifier combinations:

**Table D6-82 SSUBW, SSUBW2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.213 ST1 (vector, multiple structures)

Store multiple single-element structures from one, two, three, or four registers.

### Syntax

```
ST1 { Vt.T }, [Xn/SP] ; T1 One register  
ST1 { Vt.T, Vt2.T }, [Xn/SP] ; T1 Two registers  
ST1 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP] ; T1 Three registers  
ST1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP] ; T1 Four registers  
ST1 { Vt.T }, [Xn/SP], imm ; T1 One register, immediate offset, Post-index  
ST1 { Vt.T }, [Xn/SP], Xm ; T1 One register, register offset, Post-index  
ST1 { Vt.T, Vt2.T }, [Xn/SP], imm ; T1 Two registers, immediate offset, Post-index  
ST1 { Vt.T, Vt2.T }, [Xn/SP], Xm ; T1 Two registers, register offset, Post-index  
ST1 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], imm ; T1 Three registers, immediate offset, Post-index  
ST1 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], Xm ; T1 Three registers, register offset, Post-index  
ST1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], imm ; T1 Four registers, immediate offset, Post-index  
ST1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], Xm ; T1 Four registers, register offset, Post-index
```

Where:

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**Vt4**

Is the name of the fourth SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset:

**One register, immediate offset**

Can be one of #8 or #16.

**Two registers, immediate offset**

Can be one of #16 or #32.

**Three registers, immediate offset**

Can be one of #24 or #48.

**Four registers, immediate offset**

Can be one of #32 or #64.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

## Usage

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 and 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.

The following tables show valid specifier combinations:

**Table D6-83 ST1 (One register, immediate offset) specifier combinations**

T	imm
8B	#8
16B	#16
4H	#8
8H	#16
2S	#8
4S	#16
1D	#8
2D	#16

**Table D6-84 ST1 (Two registers, immediate offset) specifier combinations**

T	imm
8B	#16
16B	#32
4H	#16
8H	#32
2S	#16
4S	#32
1D	#16
2D	#32

**Table D6-85 ST1 (Three registers, immediate offset) specifier combinations**

T	imm
8B	#24
16B	#48
4H	#24
8H	#48
2S	#24
4S	#48

**Table D6-85 ST1 (Three registers, immediate offset) specifier combinations (continued)**

<i>T</i>	<i>imm</i>
1D	#24
2D	#48

**Table D6-86 ST1 (Four registers, immediate offset) specifier combinations**

<i>T</i>	<i>imm</i>
8B	#32
16B	#64
4H	#32
8H	#64
2S	#32
4S	#64
1D	#32
2D	#64

***Related reference***

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.214 ST1 (vector, single structure)

Store a single-element structure from one lane of one register.

### Syntax

```
ST1 { Vt.B }[index], [Xn/SP] ; T1 8-bit  
ST1 { Vt.H }[index], [Xn/SP] ; T1 16-bit  
ST1 { Vt.S }[index], [Xn/SP] ; 32-bit  
ST1 { Vt.D }[index], [Xn/SP] ; 64-bit  
ST1 { Vt.B }[index], [Xn/SP], #1 ; T1 8-bit, immediate offset, Post-index  
ST1 { Vt.B }[index], [Xn/SP], Xm ; T1 8-bit, register offset, Post-index  
ST1 { Vt.H }[index], [Xn/SP], #2 ; T1 16-bit, immediate offset, Post-index  
ST1 { Vt.H }[index], [Xn/SP], Xm ; T1 16-bit, register offset, Post-index  
ST1 { Vt.S }[index], [Xn/SP], #4 ; 32-bit, immediate offset  
ST1 { Vt.S }[index], [Xn/SP], Xm ; 32-bit, register offset  
ST1 { Vt.D }[index], [Xn/SP], #8 ; 64-bit, immediate offset  
ST1 { Vt.D }[index], [Xn/SP], Xm ; 64-bit, register offset
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**index**

The value depends on the instruction variant:

**8-bit**

Is the element index, in the range 0 to 15.

**16-bit**

Is the element index, in the range 0 to 7.

**32-bit**

Is the element index, in the range 0 to 3.

**64-bit**

Is the element index, and can be either 0 or 1.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

### Usage

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.215 ST2 (vector, multiple structures)

Store multiple 2-element structures from two registers.

### Syntax

```
ST2 { Vt.T, Vt2.T }, [Xn/SP] ; T2
ST2 { Vt.T, Vt2.T }, [Xn/SP], imm ; T2
ST2 { Vt.T, Vt2.T }, [Xn/SP], Xm ; T2
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be either #16 or #32.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.216 ST2 (vector, single structure)

Store single 2-element structure from one lane of two registers.

### Syntax

```
ST2 { Vt.B, Vt2.B }[index], [Xn/SP] ; T2
ST2 { Vt.H, Vt2.H }[index], [Xn/SP] ; T2
ST2 { Vt.S, Vt2.S }[index], [Xn/SP] ; 32-bit
ST2 { Vt.D, Vt2.D }[index], [Xn/SP] ; 64-bit
ST2 { Vt.B, Vt2.B }[index], [Xn/SP], #2 ; T2
ST2 { Vt.B, Vt2.B }[index], [Xn/SP], Xm ; T2
ST2 { Vt.H, Vt2.H }[index], [Xn/SP], #4 ; T2
ST2 { Vt.H, Vt2.H }[index], [Xn/SP], Xm ; T2
ST2 { Vt.S, Vt2.S }[index], [Xn/SP], #8 ; 32-bit, immediate offset
ST2 { Vt.S, Vt2.S }[index], [Xn/SP], Xm ; 32-bit, register offset
ST2 { Vt.D, Vt2.D }[index], [Xn/SP], #16 ; 64-bit, immediate offset
ST2 { Vt.D, Vt2.D }[index], [Xn/SP], Xm ; 64-bit, register offset
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**index**

The value depends on the instruction variant:

**8-bit**

Is the element index, in the range 0 to 15.

**16-bit**

Is the element index, in the range 0 to 7.

**32-bit**

Is the element index, in the range 0 to 3.

**64-bit**

Is the element index, and can be either 0 or 1.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

### Usage

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 and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.217 ST3 (vector, multiple structures)

Store multiple 3-element structures from three registers.

### Syntax

```
ST3 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP] ; T3
ST3 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], imm ; T3
ST3 { Vt.T, Vt2.T, Vt3.T }, [Xn/SP], Xm ; T3
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be either #24 or #48.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.218 ST3 (vector, single structure)

Store single 3-element structure from one lane of three registers.

### Syntax

```
ST3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn/SP] ; T3
ST3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn/SP] ; T3
ST3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn/SP] ; 32-bit
ST3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn/SP] ; 64-bit
ST3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn/SP], #3 ; T3
ST3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn/SP], Xm ; T3
ST3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn/SP], #6 ; T3
ST3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn/SP], Xm ; T3
ST3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn/SP], #12 ; 32-bit, immediate offset
ST3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn/SP], Xm ; 32-bit, register offset
ST3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn/SP], #24 ; 64-bit, immediate offset
ST3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn/SP], Xm ; 64-bit, register offset
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**index**

The value depends on the instruction variant:

**8-bit**

Is the element index, in the range 0 to 15.

**16-bit**

Is the element index, in the range 0 to 7.

**32-bit**

Is the element index, in the range 0 to 3.

**64-bit**

Is the element index, and can be either 0 or 1.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

### Usage

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 and 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.

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.219 ST4 (vector, multiple structures)

Store multiple 4-element structures from four registers.

### Syntax

```
ST4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP] ;
ST4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], imm ;
ST4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn/SP], Xm ;
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**Vt4**

Is the name of the fourth SIMD and FP register to be transferred.

**imm**

Is the post-index immediate offset, and can be either #32 or #64.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

### Usage

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.220 ST4 (vector, single structure)

Store single 4-element structure from one lane of four registers.

### Syntax

```
ST4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn/SP] ;
ST4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn/SP] ;
ST4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn/SP] ; 32-bit
ST4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn/SP] ; 64-bit
ST4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn/SP], #4 ;
ST4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn/SP], Xm ;
ST4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn/SP], #8 ;
ST4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn/SP], Xm ;
ST4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn/SP], #16 ; 32-bit, immediate offset
ST4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn/SP], Xm ; 32-bit, register offset
ST4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn/SP], #32 ; 64-bit, immediate offset
ST4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn/SP], Xm ; 64-bit, register offset
```

Where:

**Vt**

Is the name of the first or only SIMD and FP register to be transferred.

**Vt2**

Is the name of the second SIMD and FP register to be transferred.

**Vt3**

Is the name of the third SIMD and FP register to be transferred.

**Vt4**

Is the name of the fourth SIMD and FP register to be transferred.

**index**

The value depends on the instruction variant:

**8-bit**

Is the element index, in the range 0 to 15.

**16-bit**

Is the element index, in the range 0 to 7.

**32-bit**

Is the element index, in the range 0 to 3.

**64-bit**

Is the element index, and can be either 0 or 1.

**Xn/SP**

Is the 64-bit name of the general-purpose base register or stack pointer.

**Xm**

Is the 64-bit name of the general-purpose post-index register, excluding XZR.

### Usage

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 and 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.

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.221 SUB (vector)

Subtract (vector).

### Syntax

SUB  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Subtract (vector). This instruction subtracts each vector element in the second source SIMD and FP register from the corresponding vector element in the first source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.222 SUBHN, SUBHN2 (vector)

Subtract returning High Narrow.

### Syntax

$\text{SUBHN}\{2\} \text{ } Vd.Tb, \text{ } Vn.Ta, \text{ } Vm.Ta$

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD and FP register from the corresponding vector element in the first source SIMD and FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see [D6.141 RSUBHN, RSUBHN2 \(vector\) on page D6-1403](#).

The SUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SUBHN2 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.

The following table shows the valid specifier combinations:

**Table D6-87 SUBHN, SUBHN2 (Vector) specifier combinations**

<Q>	Tb	Ta
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.223 SUQADD (vector)

Signed saturating Accumulate of Unsigned value.

### Syntax

SUQADD  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD and FP register to corresponding signed integer values of the vector elements in the destination SIMD and FP register, and writes the resulting signed integer values to the destination SIMD and 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.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.224 SXTL, SXTL2 (vector)

Signed extend Long.

This instruction is an alias of SSHLL, SSHLL2.

The equivalent instruction is  $\text{SSHLL}\{2\} \text{ Vd.Ta, Vn.Tb, } \#0$ .

### Syntax

$\text{SXTL}\{2\} \text{ Vd.Ta, Vn.Tb}$

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD and FP register into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SXTL instruction extracts the source vector from the lower half of the source register, while the SXTL2 instruction extracts the source vector from the upper half of the 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.

The following table shows the valid specifier combinations:

**Table D6-88 SXTL, SXTL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.225 TBL (vector)

Table vector Lookup.

### Syntax

```
TBL Vd.Ta, { Vn.16B }, Vm.Ta ; Single register table
TBL Vd.Ta, { Vn.16B, <Vn+1>.16B }, Vm.Ta ; Two register table
TBL Vd.Ta, { Vn.16B, <Vn+1>.16B, <Vn+2>.16B }, Vm.Ta ; Three register table
TBL Vd.Ta, { Vn.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, Vm.Ta ; Four register table
```

Where:

**Vn**

The value depends on the instruction variant:

#### Single register table

Is the name of the SIMD and FP table register

#### Two, Three, or Four register table

Is the name of the first SIMD and FP table register

**<Vn+1>**

Is the name of the second SIMD and FP table register.

**<Vn+2>**

Is the name of the third SIMD and FP table register.

**<Vn+3>**

Is the name of the fourth SIMD and FP table register.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 8B or 16B.

**Vm**

Is the name of the SIMD and FP index register.

### Usage

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD and 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 and FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.226 TBX (vector)

Table vector lookup extension.

### Syntax

```
TBX Vd.Ta, { Vn.16B }, Vm.Ta ; Single register table
TBX Vd.Ta, { Vn.16B, <Vn+1>.16B }, Vm.Ta ; Two register table
TBX Vd.Ta, { Vn.16B, <Vn+1>.16B, <Vn+2>.16B }, Vm.Ta ; Three register table
TBX Vd.Ta, { Vn.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, Vm.Ta ; Four register table
```

Where:

**Vn**

The value depends on the instruction variant:

#### Single register table

Is the name of the SIMD and FP table register

#### Two, Three, or Four register table

Is the name of the first SIMD and FP table register

**<Vn+1>**

Is the name of the second SIMD and FP table register.

**<Vn+2>**

Is the name of the third SIMD and FP table register.

**<Vn+3>**

Is the name of the fourth SIMD and FP table register.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 8B or 16B.

**Vm**

Is the name of the SIMD and FP index register.

### Usage

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD and 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 and FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.227 TRN1 (vector)

Transpose vectors (primary).

### Syntax

TRN1 *Vd.T*, *Vn.T*, *Vm.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Usage

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD and FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD and FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

#### Note

By using this instruction with TRN2, a 2 x 2 matrix can be transposed.

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.

#### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.228 TRN2 (vector)

Transpose vectors (secondary).

### Syntax

TRN2  $Vd.T$ ,  $Vn.T$ ,  $Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD and FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD and FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

#### Note

By using this instruction with TRN1, a 2 x 2 matrix can be transposed.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.229 UABA (vector)

Unsigned Absolute difference and Accumulate.

### Syntax

UABA  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD and FP register from the corresponding elements of the first source SIMD and FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.230 UABAL, UABAL2 (vector)

Unsigned Absolute difference and Accumulate Long.

### Syntax

**UABAL{2} Vd.Ta, Vn.Tb, Vm.Tb**

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD and FP register from the corresponding vector elements of the first source SIMD and FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-89 UABAL, UABAL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.231 UABD (vector)

Unsigned Absolute Difference (vector).

### Syntax

UABD  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD and FP register from the corresponding elements of the first source SIMD and FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.232 UABDL, UABDL2 (vector)

Unsigned Absolute Difference Long.

### Syntax

`UABDL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See `<Q>` in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD and FP register from the corresponding vector elements of the first source SIMD and FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-90 UABDL, UABDL2 (Vector) specifier combinations**

<code>&lt;Q&gt;</code>	<code>Ta</code>	<code>Tb</code>
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.233 UADALP (vector)

Unsigned Add and Accumulate Long Pairwise.

### Syntax

UADALP *Vd.Ta, Vn.Tb*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD and FP register and accumulates the results with the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

Table D6-91 UADALP (Vector) specifier combinations

<i>Ta</i>	<i>Tb</i>
4H	8B
8H	16B
2S	4H
4S	8H
1D	2S
2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.234 UADDL, UADDL2 (vector)

Unsigned Add Long (vector).

### Syntax

`UADDL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See `<Q>` in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD and FP register to the corresponding vector element of the second source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and 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 UADDL instruction extracts each source vector from the lower half of each source register, while the UADDL2 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.

The following table shows the valid specifier combinations:

**Table D6-92 UADDL, UADDL2 (Vector) specifier combinations**

<code>&lt;Q&gt;</code>	<code>Ta</code>	<code>Tb</code>
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.235 UADDLP (vector)

Unsigned Add Long Pairwise.

### Syntax

UADDLP *Vd.Ta, Vn.Tb*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

Table D6-93 UADDLP (Vector) specifier combinations

<i>Ta</i>	<i>Tb</i>
4H	8B
8H	16B
2S	4H
4S	8H
1D	2S
2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.236 UADDLV (vector)

Unsigned sum Long across Vector.

### Syntax

UADDLV *Vd, Vn.T*

Where:

*V*

Is the destination width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD and FP register together, and writes the scalar result to the destination SIMD and FP register. The destination scalar is twice as long as the source vector elements. 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.

The following table shows the valid specifier combinations:

**Table D6-94 UADDLV (Vector) specifier combinations**

<i>V</i>	<i>T</i>
H	8B
H	16B
S	4H
S	8H
D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.237 UADDW, UADDW2 (vector)

Unsigned Add Wide.

### Syntax

`UADDW{2} Vd.Ta, Vn.Ta, Vm.Tb`

Where:

**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. See `<Q>` in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD and FP register to the corresponding vector elements in the lower or upper half of the second source SIMD and FP register, places the result in a vector, and writes the vector to the SIMD and FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register, while the UADDW2 instruction extracts vector elements from the upper half of the second 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.

The following table shows the valid specifier combinations:

**Table D6-95 UADDW, UADDW2 (Vector) specifier combinations**

<code>&lt;Q&gt;</code>	<code>Ta</code>	<code>Tb</code>
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.238 UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector).

### Syntax

UCVTF *Vd.T, Vn.T, #fbits*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*fbits*

Is the number of fractional bits, in the range 1 to the element width.

### Usage

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

The following table shows the valid specifier combinations:

**Table D6-96 UCVTF (Vector) specifier combinations**

<i>T</i>	<i>fbits</i>
4H	
8H	
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.239 UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector).

### Syntax

UCVTF *Vd.T, Vn.T* ; Vector half precision

UCVTF *Vd.T, Vn.T* ; Vector single-precision and double-precision

Where:

*Vd*

Is the name of the SIMD and FP destination register

*T*

Is an arrangement specifier:

#### Vector half precision

Can be one of 4H or 8H.

#### Vector single-precision and double-precision

Can be one of 2S, 4S or 2D.

*Vn*

Is the name of the SIMD and FP source register

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

### Usage

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and 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* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

### Related information

[Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#)

## D6.240 UDOT (vector, by element)

Dot Product unsigned arithmetic (vector, by element).

### Syntax

UDOT *Vd.Ta, Vn.Tb, Vm.4B[index]*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be either 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be either 8B or 16B.

*Vm*

Is the name of the second SIMD and FP source register in the range 0 to 31.

*index*

Is the element index.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

### Usage

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the 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.

---

#### Note

---

ID\_AA64ISAR0\_EL1.DP indicates whether this instruction is supported. See ID\_AA64ISAR0\_EL1 in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.241 UDOT (vector)

Dot Product unsigned arithmetic (vector).

### Syntax

UDOT *Vd.Ta, Vn.Tb, Vm.Tb*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*Ta*

Is an arrangement specifier, and can be either 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register.

*Tb*

Is an arrangement specifier, and can be either 8B or 16B.

*Vm*

Is the name of the second SIMD and FP source register.

### Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

### Usage

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the 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.

---

#### Note

---

ID\_AA64ISAR0\_EL1.DP indicates whether this instruction is supported. See *ID\_AA64ISAR0\_EL1* in the [Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile](#).

---

#### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.242 UHADD (vector)

Unsigned Halving Add.

### Syntax

UHADD  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD and FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see [D6.266 URHADD \(vector\) on page D6-1536](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.243 UHSUB (vector)

Unsigned Halving Subtract.

### Syntax

UHSUB  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD and FP register from the corresponding vector elements in the first source SIMD and FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD and 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.

### *Related reference*

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.244 UMAX (vector)

Unsigned Maximum (vector).

### Syntax

UMAX  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD and FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.245 UMAXP (vector)

Unsigned Maximum Pairwise.

### Syntax

UMAXP  $Vd.T$ ,  $Vn.T$ ,  $Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.246 UMAXV (vector)

Unsigned Maximum across Vector.

### Syntax

UMAXV *Vd, Vn.T*

Where:

*V*

Is the destination width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the largest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

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

The following table shows the valid specifier combinations:

Table D6-97 UMAXV (Vector) specifier combinations

<i>V</i>	<i>T</i>
B	8B
B	16B
H	4H
H	8H
S	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.247 UMIN (vector)

Unsigned Minimum (vector).

### Syntax

UMIN  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD and FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD and FP register.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.248 UMINP (vector)

Unsigned Minimum Pairwise.

### Syntax

UMINP  $Vd.T$ ,  $Vn.T$ ,  $Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.249 UMINV (vector)

Unsigned Minimum across Vector.

### Syntax

UMINV *Vd, Vn.T*

Where:

*V*

Is the destination width specifier, and can be one of the values shown in Usage.

*d*

Is the number of the SIMD and FP destination register.

*Vn*

Is the name of the SIMD and FP source register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the smallest of the values as a scalar to the destination SIMD and 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.

The following table shows the valid specifier combinations:

Table D6-98 UMINV (Vector) specifier combinations

<i>V</i>	<i>T</i>
B	8B
B	16B
H	4H
H	8H
S	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.250 UMLAL, UMLAL2 (vector, by element)

Unsigned Multiply-Add Long (vector, by element).

### Syntax

`UMLAL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register and accumulates the results with the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-99 UMLAL, UMLAL2 (Vector) specifier combinations**

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.251 UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector).

### Syntax

`UMLAL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See `<Q>` in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD and FP register by the corresponding vector elements of the second source SIMD and FP register, and accumulates the results with the vector elements of the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-100 UMLAL, UMLAL2 (Vector) specifier combinations**

<code>&lt;Q&gt;</code>	<code>Ta</code>	<code>Tb</code>
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.252 UMLSL, UMLSL2 (vector, by element)

Unsigned Multiply-Subtract Long (vector, by element).

### Syntax

`UMLSL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register and subtracts the results from the vector elements of the destination SIMD and 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\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

**Table D6-101 UMLSL, UMLSL2 (Vector) specifier combinations**

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.253 UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector).

### Syntax

`UMLSL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD and FP registers, and subtracts the results from the vector elements of the destination SIMD and 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\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

**Table D6-102 UMLSL, UMLSL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.254 UMOV (vector)

Unsigned Move vector element to general-purpose register.

This instruction is used by the alias **MOV** (to general).

### Syntax

`UMOV Wd, Vn.Ts[index] ; 32-bit`

`UMOV Xd, Vn.Ts[index] ; 64-bit`

Where:

**Wd**

Is the 32-bit name of the general-purpose destination register.

**Ts**

Is an element size specifier:

**32-bit**

Can be one of B, H or S.

**64-bit**

Must be D.

**index**

The value depends on the instruction variant:

**32-bit**

Is the element index, in the range shown in Usage.

**64-bit**

Is the element index and can be either 0 or 1.

**Xd**

Is the 64-bit name of the general-purpose destination register.

**Vn**

Is the name of the SIMD and FP source register.

### Usage

Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD and FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose 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.

**Table D6-103 UMOV (32-bit) specifier combinations**

Ts	index
B	0 to 15
H	0 to 7
S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.255 UMULL, UMULL2 (vector, by element)

Unsigned Multiply Long (vector, by element).

### Syntax

`UMULL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be either 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

**Ts**

Is an element size specifier, and can be either H or S.

**index**

Is the element index, in the range shown in Usage.

### Usage

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and 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.

The following table shows the valid specifier combinations:

**Table D6-104 UMULL, UMULL2 (Vector) specifier combinations**

<Q>	Ta	Tb	Ts	index
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.256 UMULL, UMULL2 (vector)

Unsigned Multiply long (vector).

### Syntax

`UMULL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD and FP registers, places the result in a vector, and writes the vector to the destination SIMD and 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\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

**Table D6-105 UMULL, UMULL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.257 UQADD (vector)

Unsigned saturating Add.

### Syntax

UQADD  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.258 UQRSHL (vector)

Unsigned saturating Rounding Shift Left (register).

### Syntax

UQRSHL  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD and FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see [D6.260 UQSHL \(vector, immediate\) on page D6-1529](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.259 UQRSHRN, UQRSHRN2 (vector)

Unsigned saturating Rounded Shift Right Narrow (immediate).

### Syntax

`UQRSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

### Usage

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [D6.262 UQSHRN, UQSHRN2 \(vector\) on page D6-1531](#).

The UQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D6-106 UQRSHRN{2} (Vector) specifier combinations**

<Q>	Tb	Ta	shift
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.260 UQSHL (vector, immediate)

Unsigned saturating Shift Left (immediate).

### Syntax

`UQSHL Vd.T, Vn.T, #shift`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**shift**

Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD and FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see [D6.258 UQRSHL \(vector\) on page D6-1527](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D6-107 UQSHL (Vector) specifier combinations**

<b>T</b>	<b>shift</b>
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.261 UQSHL (vector, register)

Unsigned saturating Shift Left (register).

### Syntax

`UQSHL Vd.T, Vn.T, Vm.T`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see [D6.258 UQRSHL \(vector\) on page D6-1527](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.262 UQSHRN, UQSHRN2 (vector)

Unsigned saturating Shift Right Narrow (immediate).

### Syntax

`UQSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

**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. See `<Q>` in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

### Usage

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [D6.259 UQRSHRN, UQRSHRN2 \(vector\) on page D6-1528](#).

The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

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.

The following table shows the valid specifier combinations:

**Table D6-108 UQSHRN{2} (Vector) specifier combinations**

<code>&lt;Q&gt;</code>	<code>Tb</code>	<code>Ta</code>	<code>shift</code>
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

**Related reference**

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.263 UQSUB (vector)

Unsigned saturating Subtract.

### Syntax

UQSUB  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD and FP register from the corresponding element values of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and 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.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.264 UQXTN, UQXTN2 (vector)

Unsigned saturating extract Narrow.

### Syntax

$UQXTN\{2\} \ Vd.Tb, \ Vn.Ta$

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD and FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The UQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQXTN2 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.

The following table shows the valid specifier combinations:

**Table D6-109 UQXTN{2} (Vector) specifier combinations**

<Q>	Tb	Ta
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.265 URECPE (vector)

Unsigned Reciprocal Estimate.

### Syntax

URECPE *Vd.T, Vn.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be either 2S or 4S.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD and FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD and FP register.

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

### Related reference

*D6.1 A64 SIMD Vector instructions in alphabetical order* on page D6-1243

## D6.266 URHADD (vector)

Unsigned Rounding Halving Add.

### Syntax

URHADD *Vd.T, Vn.T,Vm.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S or 4S.

*Vn*

Is the name of the first SIMD and FP source register.

*Vm*

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD and FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see [D6.242 UHADD \(vector\) on page D6-1511](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.267 URSHL (vector)

Unsigned Rounding Shift Left (register).

### Syntax

URSHL  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

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

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.268 URSHR (vector)

Unsigned Rounding Shift Right (immediate).

### Syntax

URSHR *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [D6.273 USHR \(vector\) on page D6-1543](#).

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.

The following table shows the valid specifier combinations:

**Table D6-110 URSHR (Vector) specifier combinations**

<i>T</i>	<i>shift</i>
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.269 URSQRTE (vector)

Unsigned Reciprocal Square Root Estimate.

### Syntax

URSQRTE *Vd.T, Vn.T*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be either 2S or 4S.

*Vn*

Is the name of the SIMD and FP source register.

### Usage

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD and FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD and 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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.270 URSRA (vector)

Unsigned Rounding Shift Right and Accumulate (immediate).

### Syntax

URSRA  $Vd.T, Vn.T, \#shift$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of the values shown in Usage.

$Vn$

Is the name of the SIMD and FP source register.

$shift$

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [D6.275 USRA \(vector\) on page D6-1545](#).

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.

The following table shows the valid specifier combinations:

**Table D6-111 URSRA (Vector) specifier combinations**

$T$	$shift$
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.271 USHL (vector)

Unsigned Shift Left (register).

### Syntax

USHL  $Vd.T, Vn.T, Vm.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the first SIMD and FP source register.

$Vm$

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see [D6.267 URSHL \(vector\) on page D6-1537](#).

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.272 USHLL, USHLL2 (vector)

Unsigned Shift Left Long (immediate).

This instruction is used by the alias UXTL, UXTL2, UXTL, UXTL22.

### Syntax

`USHLL{2} Vd.Ta, Vn.Tb, #shift`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**shift**

Is the left shift amount, in the range 0 to the source element width in bits minus 1, and can be one of the values shown in Usage.

### Usage

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD and FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

The USHLL instruction extracts vector elements from the lower half of the source register, while the USHLL2 instruction extracts vector elements from the upper half of the 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.

The following table shows the valid specifier combinations:

Table D6-112 USHLL, USHLL2 (Vector) specifier combinations

<Q>	Ta	Tb	shift
-	8H	8B	0 to 7
2	8H	16B	0 to 7
-	4S	4H	0 to 15
2	4S	8H	0 to 15
-	2D	2S	0 to 31
2	2D	4S	0 to 31

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.273 USHR (vector)

Unsigned Shift Right (immediate).

### Syntax

USHR *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [D6.268 URSHR \(vector\) on page D6-1538](#).

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.

The following table shows the valid specifier combinations:

**Table D6-113 USHR (Vector) specifier combinations**

<i>T</i>	<i>shift</i>
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.274 USQADD (vector)

Unsigned saturating Accumulate of Signed value.

### Syntax

USQADD  $Vd.T, Vn.T$

Where:

$Vd$

Is the name of the SIMD and FP destination register.

$T$

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

$Vn$

Is the name of the SIMD and FP source register.

### Usage

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD and FP register to corresponding unsigned integer values of the vector elements in the destination SIMD and FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD and 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.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.275 USRA (vector)

Unsigned Shift Right and Accumulate (immediate).

### Syntax

USRA *Vd.T, Vn.T, #shift*

Where:

*Vd*

Is the name of the SIMD and FP destination register.

*T*

Is an arrangement specifier, and can be one of the values shown in Usage.

*Vn*

Is the name of the SIMD and FP source register.

*shift*

Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

### Usage

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [D6.270 URSRA \(vector\) on page D6-1540](#).

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.

The following table shows the valid specifier combinations:

Table D6-114 USRA (Vector) specifier combinations

<i>T</i>	<i>shift</i>
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order on page D6-1243](#)

## D6.276 USUBL, USUBL2 (vector)

Unsigned Subtract Long.

### Syntax

USUBL{2}  $Vd.Ta, Vn.Tb, Vm.Tb$

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD and FP register from the corresponding vector element of the first source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register, while the USUBL2 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.

The following table shows the valid specifier combinations:

**Table D6-115 USUBL, USUBL2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.277 USUBW, USUBW2 (vector)

Unsigned Subtract Wide.

### Syntax

**USUBW{2} Vd.Ta, Vn.Ta, Vm.Tb**

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD and FP register from the corresponding vector element in the lower or upper half of the first source SIMD and FP register, places the result in a vector, and writes the vector to the SIMD and FP destination register. All the values in this instruction are signed integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source register, while the USUBW2 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.

The following table shows the valid specifier combinations:

**Table D6-116 USUBW, USUBW2 (Vector) specifier combinations**

<Q>	Ta	Tb
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.278 UXTL, UXTL2 (vector)

Unsigned extend Long.

This instruction is an alias of USHLL, USHLL2.

The equivalent instruction is USHLL{2}  $Vd.Ta, Vn.Tb, \#0$ .

### Syntax

$UXTL\{2\} Vd.Ta, Vn.Tb$

Where:

$2$

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. See  $\langle Q \rangle$  in the Usage table.

$Vd$

Is the name of the SIMD and FP destination register.

$Ta$

Is an arrangement specifier, and can be one of the values shown in Usage.

$Vn$

Is the name of the SIMD and FP source register.

$Tb$

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD and FP register into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

The UXTL instruction extracts vector elements from the lower half of the source register, while the UXTL2 instruction extracts vector elements from the upper half of the 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.

The following table shows the valid specifier combinations:

**Table D6-117 UXTL, UXTL2 (Vector) specifier combinations**

$\langle Q \rangle$	$Ta$	$Tb$
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.279 UZP1 (vector)

Unzip vectors (primary).

### Syntax

`UZP1 Vd.T, Vn.T, Vm.T`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD and FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD and FP register.

————— Note —————

This instruction can be used with UZP2 to de-interleave two vectors.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.280 UZP2 (vector)

Unzip vectors (secondary).

### Syntax

`UZP2 Vd.T, Vn.T, Vm.T`

Where:

**Vd**

Is the name of the SIMD and FP destination register.

**T**

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

**Vn**

Is the name of the first SIMD and FP source register.

**Vm**

Is the name of the second SIMD and FP source register.

### Usage

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD and FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD and FP register.

————— Note —————

This instruction can be used with `UZP1` to de-interleave two vectors.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.281 XTN, XTN2 (vector)

Extract Narrow.

### Syntax

`XTN{2} Vd.Tb, Vn.Ta`

Where:

**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. See <Q> in the Usage table.

**Vd**

Is the name of the SIMD and FP destination register.

**Tb**

Is an arrangement specifier, and can be one of the values shown in Usage.

**Vn**

Is the name of the SIMD and FP source register.

**Ta**

Is an arrangement specifier, and can be one of the values shown in Usage.

### Usage

Extract Narrow. This instruction reads each vector element from the source SIMD and FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements.

The **XTN** instruction writes the vector to the lower half of the destination register and clears the upper half, while the **XTN2** 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.

The following table shows the valid specifier combinations:

**Table D6-118 XTN, XTN2 (Vector) specifier combinations**

<Q>	Tb	Ta
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.282 ZIP1 (vector)

Zip vectors (primary).

### Syntax

**ZIP1** *Vd.T, Vn.T,Vm.T*

Where:

***Vd***

Is the name of the SIMD and FP destination register.

***T***

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

***Vn***

Is the name of the first SIMD and FP source register.

***Vm***

Is the name of the second SIMD and FP source register.

### Usage

Zip vectors (primary). This instruction reads adjacent vector elements from the upper half of two source SIMD and FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD and FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

---

— **Note** —

This instruction can be used with **ZIP2** to interleave two vectors.

---

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243

## D6.283 ZIP2 (vector)

Zip vectors (secondary).

### Syntax

**ZIP2 *Vd.T, Vn.T,Vm.T***

Where:

***Vd***

Is the name of the SIMD and FP destination register.

***T***

Is an arrangement specifier, and can be one of 8B, 16B, 4H, 8H, 2S, 4S or 2D.

***Vn***

Is the name of the first SIMD and FP source register.

***Vm***

Is the name of the second SIMD and FP source register.

### Usage

Zip vectors (secondary). This instruction reads adjacent vector elements from the lower half of two source SIMD and FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD and FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

————— Note —————

This instruction can be used with **ZIP1** to interleave two vectors.

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.

### Related reference

[D6.1 A64 SIMD Vector instructions in alphabetical order](#) on page D6-1243



# Chapter D7

## A64 Cryptographic Algorithms

Lists the algorithms that A64 SIMD instructions support.

It contains the following section:

- [\*D7.1 A64 Cryptographic instructions\* on page D7-1556.](#)

## D7.1 A64 Cryptographic instructions

A set of A64 cryptographic instructions is available in the Armv8 architecture.

These instructions use the 128-bit Advanced SIMD registers and support the acceleration of the following cryptographic and hash algorithms:

- AES.
- SHA1.
- SHA256.
- SHA3, optional in architectures Armv8.2-A and later.
- SHA512, optional in architectures Armv8.2-A and later.
- SM3, optional in architectures Armv8.2-A and later.
- SM4, optional in architectures Armv8.2-A and later.

### Summary of A64 cryptographic instructions

The following table lists the A64 cryptographic instructions that are supported:

**Table D7-1 Summary of A64 cryptographic instructions**

Mnemonic	Brief description
AESD	AES single round decryption
AESE	AES single round encryption
AESIMC	AES inverse mix columns
AESMC	AES mix columns
BCAX	SHA3 Bit Clear and XOR
EOR3	SHA3 Three-way Exclusive OR
RAX1	SHA3 Rotate and Exclusive OR
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
SHA256H2	SHA256 hash update (part 2)
SHA256H	SHA256 hash update (part 1)
SHA256SU0	SHA256 schedule update 0
SHA256SU1	SHA256 schedule update 1
SHA512H2	SHA512 Hash update part 2
SHA512H	SHA512 Hash update part 1
SHA512SU0	SHA512 Schedule Update 0
SHA512SU1	SHA512 Schedule Update 1
SM3PARTW1	SM3 three-way exclusive OR on the combination of three 128-bit vectors
SM3PARTW2	SM3 three-way exclusive OR on the combination of three 128-bit vectors

**Table D7-1 Summary of A64 cryptographic instructions (continued)**

Mnemonic	Brief description
SM3SS1	SM3 perform rotates and adds on three 128-bit vectors combined into a destination 128-bit SIMD and FP register
SM3TT1A	SM3 three-way exclusive OR on the combination of three 128-bit vectors and a 2-bit immediate index value
SM3TT1B	SM3 perform 32-bit majority function on the combination of three 128-bit vectors and 2-bit immediate index value
SM3TT2A	SM3 three-way exclusive OR of combined three 128-bit vectors and a 2-bit immediate index value
SM3TT2B	SM3 perform 32-bit majority function on the combination of three 128-bit vectors and 2-bit immediate index value
SM4E	SM4 Encode
SM4EKEY	SM4 Key
XAR	SHA3 Exclusive OR and Rotate

