Arm® Architecture Reference Manual

for A-profile architecture

Document number	ARM DDI 0487
Document quality	EAC
Document version	M.a.a
Document confidentiality	Non-Confidential

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.

Release information

The following releases of this document have been made.

Date	Version	Changes
17/Dec/2025	M.a.a	• Non-Confidential Armv9.6 EAC release with event links fix. No other content changes.
01/Dec/2025	M.a	• Initial Non-Confidential Armv9.6 EAC release
30/Apr/2025	L.b	• Maintenance release with the MPAM PE content re-written as rules and the 2025-03 release of the Features, ISA, and System registers content
30/Nov/2024	L.a	• Initial Non-Confidential Armv9.5 EAC release and separation of the RAS System Architecture into ARM IHI 0100
20/Mar/2024	K.a	• Initial Non-Confidential Armv8.9 and Armv9.4 EAC release, incorporating MPAM and SME
21/Apr/2023	J.a	• Updated Non-Confidential Armv8.8 and Armv9.3 EAC release incorporating MEC, RAS, and RME
19/Aug/2022	I.a	• Updated Non-Confidential Armv8.8 and Armv9.3 EAC release incorporating BRBE, ETE, and TRBE
04/Feb/2022	H.a	• Initial Non-Confidential Armv8.8 and Armv9.3 EAC release, incorporating SVE
22/Jul/2021	G.b	• Updated Non-Confidential Armv8.7 EAC release
22/Jan/2021	G.a	• Initial Non-Confidential Armv8.7 EAC release
17/Jul/2020	F.c	• Initial Non-Confidential Armv8.6 EAC release
31/Mar/2020	F.b	• Non-Confidential Armv8.5 EAC, initial Armv8.6 Beta release
20/Feb/2020	F.a	• Initial Non-Confidential Armv8.6 Beta release
05/Jul/2019	E.a	• Initial Non-Confidential Armv8.5 EAC release
29/Apr/2019	D.b	• Updated Non-Confidential Armv8.4 EAC release incorporating accessibility changes
31/Oct/2018	D.a	• Initial Non-Confidential Armv8.4 EAC release
20/Dec/2017	C.a	• Initial Non-Confidential Armv8.3 EAC release
26/Sep/2017	B.b	• Initial Non-Confidential Armv8.2 EAC release, incorporating SPE
31/Mar/2017	B.a	• Initial Non-Confidential Armv8.1 EAC, v8.2 Beta release
30/Sep/2016	A.k	• Updated Non-Confidential EAC release
03/Jun/2016	A.j	• Non-Confidential EAC release
28/Jan/2016	A.i	• Ninth Non-Confidential Beta release
30/Sep/2015	A.h	• Eighth Non-Confidential Beta release
10/Jul/2015	A.g	• Seventh Non-Confidential Beta release
25/Mar/2015	A.f	• Sixth Non-Confidential Beta release
17/Dec/2014	A.e	• Fifth Non-Confidential Beta release
09/Oct/2014	A.d	• Fourth Non-Confidential Beta release
18/Jul/2014	A.c	• Third Non-Confidential Beta release
24/Dec/2013	A.b	• Second Non-Confidential Beta release
04/Sep/2013	A.a	• Non-Confidential Beta release
12/Jun/2013	A.a-2	• Second confidential Beta draft of first issue, limited circulation
30/Apr/2013	A.a-1	• Confidential Beta draft of first issue, limited circulation

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Non-Confidential Proprietary Notice

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

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

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. 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, any patents, copyrights, trade secrets, trademarks, or other rights.

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.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted 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.

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 this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

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

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-20349

8 March 2024

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.

Product Status

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

The information in this manual is at EAC quality, which means that all features of the specification are described in the manual.

Web Address

https://www.arm.com

Limitations of this issue of the Manual

This issue of the Arm Architecture Reference Manual contains many improvements and corrections.

Validation of this document has identified the following issues that Arm will address in future issues:

- Some diagrams in the register descriptions chapter have long field names that split over several lines. These are not wrong, and the descriptions list the field name correctly.
- Arm Pseudocode Definition requires further review and update. Since this appendix is informative, rather than being part of the architecture specification, this does not affect the quality status of this release.
- For a list of the known issues in this Manual, please refer to the Known Issues document on Developer.
- For a list of the known issues in the most recent Register and Instruction XML releases, please refer to the Release Notes on Developer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

Arm® Architecture Reference Manual, for A-profile architecture

Preface

About this Manual 23

Using this Manual 25

Conventions 32

Additional reading 35

Feedback 36

Part A — **Arm Architecture Introduction and Overview**

Chapter A1 — **Introduction to the Arm Architecture**

A1.1 About the Arm architecture 39

A1.2 Architecture profiles 41

A1.3 Arm architectural concepts 42

A1.4 Supported data types 45

A1.5 Floating-point support 61

A1.6 The Arm memory model 79

A1.7 Reliability, Availability, and Serviceability 80

Chapter A2 — **A-profile Architecture Extensions**

A2.1 About the A-profile architecture extensions 84

A2.2 Armv8-A architecture extensions 85

A2.3 Armv9-A architecture extensions 161

Part B — **The AArch64 Application Level Architecture**

Chapter B1 — **The AArch64 Application Level Programmers' Model**

B1.1 About the Application level programmers' model 202

B1.2 Registers in AArch64 Execution state 203

B1.3 Process state, PSTATE 207

B1.4 The Scalable Vector and Scalable Matrix Extensions (SVE & SME) 210

B1.5 Software control features and EL0 239

Chapter B2 — **The AArch64 Application Level Memory Model**

B2.1 About the Arm memory model 244

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.2 Atomicity in the Arm architecture 246
B2.3 Ordering requirements defined by the formal concurrency model 251
B2.4 Additional ordering requirements outside of the scope of the formal concurrency model 283
B2.5 Restrictions on the effects of speculation 288
B2.6 Memory barriers 295
B2.7 Caches and memory hierarchy 304
B2.8 Alignment support 309
B2.9 Endian support 312
B2.10 Memory types and attributes 316
B2.11 Mismatched memory attributes 328
B2.12 Synchronization and semaphores 331

Part C The AArch64 Instruction Set

Chapter C1 The A64 Instruction Set

C1.1 About the A64 instruction set 344
C1.2 Structure of the A64 assembler language 345
C1.3 Address generation 351
C1.4 Instruction aliases 354

Chapter C2 About the A64 Instruction Descriptions

C2.1 Understanding the A64 instruction descriptions 356
C2.2 General information about the A64 instruction descriptions 359

Chapter C3 A64 Instruction Set Overview

C3.1 Branches, Exception generating, and System instructions 363
C3.2 Loads and stores 376
C3.3 Loads and stores - SVE, SVE2 408
C3.4 Loads and stores - SME, SME2, SVE2p1 418
C3.5 Data processing - immediate 423
C3.6 Data Processing - MTE 429
C3.7 Data processing - register 430
C3.8 Data processing - SIMD and floating-point 439
C3.9 Data processing - SVE 468
C3.10 Data processing - SVE2 493
C3.11 Data processing - SME, SME2 521

Chapter C4 A64 Instruction Set Encoding

C4.1 A64 instruction set encoding 543

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C5 **The A64 System Instruction Class**

C5.1 The System instruction class encoding space 915

C5.2 Special-purpose registers 935

C5.3 A64 System instructions for cache maintenance 1065

C5.4 A64 System instructions for address translation 1152

C5.5 A64 System instructions for TLB maintenance 1184

C5.6 A64 System instructions for prediction restriction 1745

C5.7 A64 System instructions for the Branch Record Buffer Extension 1762

C5.8 A64 System instructions for the Trace Extension 1767

C5.9 A64 System instructions for the Guarded Control Stack 1769

C5.10 A64 IMPLEMENTATION DEFINED System instructions 1779

Chapter C6 **A64 Base Instruction Descriptions**

C6.1 About the A64 base instructions 1784

C6.2 Alphabetical list of A64 base instructions 1787

Chapter C7 **A64 Advanced SIMD and Floating-point Instruction Descriptions**

C7.1 About the A64 Advanced SIMD and floating-point instructions 2874

C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions 2877

Chapter C8 **SVE Instruction Descriptions**

C8.1 About the SVE instructions 4034

C8.2 Alphabetical list of SVE instructions 4035

Chapter C9 **SME Instruction Descriptions**

C9.1 About the SME instructions 6129

C9.2 Alphabetical list of SME instructions 6130

Part D The AArch64 System Level Architecture

Chapter D1 **The AArch64 System Level Programmers' Model**

D1.1 Exception levels 7081

D1.2 Registers for instruction processing and exception handling 7086

D1.3 System register locking 7088

D1.4 Exceptions 7090

D1.5 Process state, PSTATE 7136

D1.6 Resets and power domains 7139

D1.7 Mechanisms for entering a low-power state 7142

D1.8 Self-hosted debug 7147

D1.9 Event monitors 7148

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.10 Interprocessing 7149
D1.11 Check Feature 7155

Chapter D2 AArch64 Self-hosted Debug

D2.1 About self-hosted debug 7157
D2.2 Routing debug exceptions 7161
D2.3 The debug exception enable controls 7163
D2.4 The effect of powerdown on debug exceptions 7165
D2.5 Summary of the routing and enabling of debug exceptions 7166
D2.6 Pseudocode description of debug exceptions 7168
D2.7 Breakpoint Instruction exceptions 7169
D2.8 Breakpoint exceptions 7171
D2.9 Watchpoint exceptions 7194
D2.10 Vector Catch exceptions 7212
D2.11 Software Step exceptions 7213
D2.12 Synchronization and debug exceptions 7228

Chapter D3 AArch64 Self-hosted Trace

D3.1 About self-hosted trace 7230
D3.2 Prohibited regions in self-hosted trace 7231
D3.3 Self-hosted trace timestamps 7233
D3.4 Synchronization in self-hosted trace 7234

Chapter D4 The Embedded Trace Extension

D4.1 About the Embedded Trace Extension 7236
D4.2 Programmers' model 7243
D4.3 Trace elements 7247
D4.4 Instruction and exception classification 7263
D4.5 About the ETE trace unit 7272
D4.6 Resource operation 7316

Chapter D5 ETE Protocol Descriptions

D5.1 About the ETE protocol 7347
D5.2 Summary list of ETE packets 7349
D5.3 Alphabetical list of ETE packets 7352

Chapter D6 The Trace Buffer Extension

D6.1 About the Trace Buffer Extension 7477
D6.2 The trace buffer 7480
D6.3 Trace buffer Self-hosted mode 7485

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.4 Trace buffer External mode 7493
D6.5 Trace buffer management 7498
D6.6 Synchronization and the Trace Buffer Unit 7513
D6.7 Trace synchronization and memory barriers 7517
D6.8 Trace of Speculative execution 7518
D6.9 Trace in Debug state 7519
D6.10 Synchronization litmus tests 7520
D6.11 UNPREDICTABLE behavior 7524

Chapter D7 The AArch64 System Level Memory Model

D7.1 About the memory system architecture 7527
D7.2 Address space 7528
D7.3 Mixed-endian support in AArch64 7529
D7.4 Memory Encryption Contexts 7530
D7.5 Cache support 7532
D7.6 External aborts 7562
D7.7 Memory barrier instructions 7564
D7.8 Pseudocode description of general memory System instructions 7565

Chapter D8 The AArch64 Virtual Memory System Architecture

D8.1 Address translation 7569
D8.2 Translation process 7585
D8.3 Translation table descriptor formats 7623
D8.4 Memory access control 7643
D8.5 Hardware updates to the translation tables 7669
D8.6 Memory region attributes 7684
D8.7 Other descriptor fields 7693
D8.8 Address tagging 7698
D8.9 Logical Address Tagging 7700
D8.10 Pointer authentication 7702
D8.11 Checked Pointer Arithmetic 7708
D8.12 Memory Encryption Contexts extension 7710
D8.13 Virtualization Host Extensions 7713
D8.14 Nested virtualization 7720
D8.15 Memory aborts 7728
D8.16 Translation Lookaside Buffers 7741
D8.17 TLB maintenance 7746
D8.18 Caches 7765

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.19 Pseudocode description of VMSA address translation 7767

Chapter D9 The Granule Protection Check Mechanism

D9.1 GPC behavior overview 7770
D9.2 GPC bypass windows 7771
D9.3 Granular Data Isolation 7772
D9.4 GPC faults 7773
D9.5 GPT caching and invalidation 7775
D9.6 GPT formats 7777
D9.7 GPT lookup process 7782
D9.8 Explicit PAS control 7784

Chapter D10 The Memory Tagging Extension

D10.1 Introduction 7786
D10.2 Allocation Tags 7787
D10.3 Memory region tagging types 7788
D10.4 Tag checking 7790
D10.5 Allocation Tag Access controls 7792
D10.6 Physical Tag locations 7793
D10.7 Tag Check Faults 7795

Chapter D11 The Guarded Control Stack

D11.1 Introduction 7798
D11.2 The Guarded Control Stack 7799
D11.3 Procedure returns 7800
D11.4 Exception returns 7803
D11.5 Stage 1 permission model 7807
D11.6 Stage 2 Permission model 7809
D11.7 Guarded Control Stack switching 7810
D11.8 Guarded Control Stack exceptions 7813
D11.9 Guarded Control Stack data accesses 7814
D11.10 Detecting when FEAT_GCS is enabled 7820

Chapter D12 The Generic Timer in AArch64 state

D12.1 About the Generic Timer 7822
D12.2 The AArch64 view of the Generic Timer 7827

Chapter D13 The Performance Monitors Extension

D13.1 About the Performance Monitors 7836
D13.2 Accuracy of the Performance Monitors 7840

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.3 Behavior on overflow 7843
D13.4 Attributability 7850
D13.5 Controlling the PMU counters 7852
D13.6 Multithreaded implementations 7857
D13.7 Event filtering 7859
D13.8 Event threshold and edge counting 7861
D13.9 PMU snapshots 7866
D13.10 Performance Monitors and Debug state 7870
D13.11 Counter access 7871
D13.12 Performance Monitors Extension registers 7873

Chapter D14 PMU Event Descriptions

D14.1 Definitions 7875
D14.2 The PMU event number space and common events 7887
D14.3 Common event numbers 7941
D14.4 Cycle event counting 8143
D14.5 Meaningful combinations of common events 8145
D14.6 Required events 8150
D14.7 Synchronous events 8152
D14.8 IMPLEMENTATION DEFINED event numbers 8158

Chapter D15 The System Performance Monitors Extension

D15.1 About the System Performance Monitors 8160
D15.2 System PMU configuration 8161
D15.3 Accessing System PMUs 8163
D15.4 Generating System PMU overflow interrupt requests 8165

Chapter D16 The Activity Monitors Extension

D16.1 About the Activity Monitors Extension 8167
D16.2 Properties and behavior of the activity monitors 8168
D16.3 AMU events and event numbers 8170

Chapter D17 The Statistical Profiling Extension

D17.1 About the Statistical Profiling Extension 8172
D17.2 Defining the sample population 8174
D17.3 Controlling when an operation is sampled 8175
D17.4 Enabling profiling 8178
D17.5 Filtering sample records 8180
D17.6 The profiling data 8183
D17.7 The Profiling Buffer 8197

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.8 Profiling Buffer management 8204
D17.9 Synchronization and Statistical Profiling 8215

Chapter D18 Statistical Profiling Extension Sample Record Specification

D18.1 About the Statistical Profiling Extension sample records 8218
D18.2 Alphabetical list of Statistical Profiling Extension packets 8221

Chapter D19 The Branch Record Buffer Extension

D19.1 About the Branch Record Buffer Extension 8263
D19.2 Branch record filtering 8268
D19.3 Branch record buffer operation 8273
D19.4 Branch record buffer 8276
D19.5 Programmers' model 8278

Chapter D20 RAS PE Architecture

D20.1 About the RAS Extension 8281
D20.2 PE error handling 8283
D20.3 Generating error exceptions 8288
D20.4 Taking error exceptions 8289
D20.5 Error synchronization event 8298
D20.6 Virtual SError exceptions and delegated SError exceptions 8304
D20.7 Error records in the PE 8305

Chapter D21 MPAM PE Architecture

D21.1 About MPAM 8310
D21.2 PARTID spaces 8314
D21.3 Partition Identifiers (PARTIDs) 8318
D21.4 Performance Monitoring Groups (PMGs) 8321
D21.5 Effect of the MPAM3_EL3.{SDEFLT, FORCE_NS} controls on PE Secure state execution 8322
D21.6 Effect of access type on MPAM information 8324
D21.7 Virtualization support 8326
D21.8 PE-side Maximum-bandwidth limit injection control 8329
D21.9 MPAM errors 8330
D21.10 Synchronization of MPAM System register changes 8331

Chapter D22 The Scalable Matrix Extension

D22.1 Overview 8333
D22.2 SME traps and exceptions 8334
D22.3 Validity of SME and SVE state 8335
D22.4 Streaming execution priority 8336

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.5 Floating-point behaviors in Streaming SVE mode 8338
D22.6 Floating-point behaviors for instructions that target the SME ZA array 8339
D22.7 Security and power considerations 8340

Chapter D23 AArch64 System Register Encoding

D23.1 The System register encoding space 8342
D23.2 Moves to and from debug and trace System registers 8343
D23.3 Moves to and from non-debug System registers, Special-purpose registers 8348

Chapter D24 AArch64 System Register Descriptions

D24.1 About the AArch64 System registers 8366
D24.2 General system control registers 8375
D24.3 Debug registers 10019
D24.4 Trace registers 10183
D24.5 Performance Monitors registers 10519
D24.6 Activity Monitors registers 10760
D24.7 Statistical Profiling Extension registers 10805
D24.8 Branch Record Buffer Extension registers 10941
D24.9 RAS registers 10991
D24.10 Generic Timer registers 11049
D24.11 Guarded Control Stack registers 11180
D24.12 MPAM registers 11211

Part E The AArch32 Application Level Architecture

Chapter E1 The AArch32 Application Level Programmers' Model

E1.1 About the Application level programmers' model 11313
E1.2 The Application level programmers' model in AArch32 state 11314
E1.3 Advanced SIMD and floating-point instructions 11327
E1.4 About the AArch32 System register interface 11345
E1.5 Exceptions 11346

Chapter E2 The AArch32 Application Level Memory Model

E2.1 About the Arm memory model 11348
E2.2 Atomicity in the Arm architecture 11350
E2.3 Definition of the memory model 11354
E2.4 Ordering of translation table walks 11365
E2.5 Caches and memory hierarchy 11366
E2.6 Alignment support 11371
E2.7 Endian support 11373

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.8 Memory types and attributes 11377
E2.9 Mismatched memory attributes 11387
E2.10 Synchronization and semaphores 11390

Part F The AArch32 Instruction Sets

Chapter F1 About the T32 and A32 Instruction Descriptions

F1.1 Format of instruction descriptions 11402
F1.2 Standard assembler syntax fields 11405
F1.3 Conditional execution 11406
F1.4 Shifts applied to a register 11408
F1.5 Memory accesses 11410
F1.6 Encoding of lists of general-purpose registers and the PC 11411
F1.7 General information about the T32 and A32 instruction descriptions 11412
F1.8 Additional pseudocode support for instruction descriptions 11426
F1.9 Additional information about Advanced SIMD and floating-point instructions 11427

Chapter F2 The AArch32 Instruction Sets Overview

F2.1 Support for instructions in different versions of the Arm architecture 11434
F2.2 Unified Assembler Language 11435
F2.3 Branch instructions 11437
F2.4 Data-processing instructions 11438
F2.5 PSTATE and banked register access instructions 11446
F2.6 Load/store instructions 11447
F2.7 Load/store multiple instructions 11450
F2.8 Miscellaneous instructions 11451
F2.9 Exception-generating and exception-handling instructions 11452
F2.10 System register access instructions 11454
F2.11 Advanced SIMD and floating-point load/store instructions 11455
F2.12 Advanced SIMD and floating-point register transfer instructions 11457
F2.13 Advanced SIMD data-processing instructions 11458
F2.14 Floating-point data-processing instructions 11469

Chapter F3 T32 Instruction Set Encoding

F3.1 T32 instruction set encoding 11472
F3.2 About the T32 Advanced SIMD and floating-point instructions and their encoding 11549

Chapter F4 A32 Instruction Set Encoding

F4.1 A32 instruction set encoding 11551
F4.2 About the A32 Advanced SIMD and floating-point instructions and their encoding 11618

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F5 **T32 and A32 Base Instruction Set Instruction Descriptions**

F5.1 Alphabetical list of T32 and A32 base instruction set instructions 11620

F5.2 Encoding and use of banked register transfer instructions 12400

Chapter F6 **T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions**

F6.1 Alphabetical list of Advanced SIMD and floating-point instructions 12405

Part G The AArch32 System Level Architecture

Chapter G1 **The AArch32 System Level Programmers' Model**

G1.1 About the AArch32 System level programmers' model 13257

G1.2 Exception levels 13258

G1.3 Exception terminology 13259

G1.4 Execution state 13261

G1.5 Instruction Set state 13263

G1.6 Security state 13264

G1.7 Security state, Exception levels, and AArch32 execution privilege 13267

G1.8 Virtualization 13269

G1.9 AArch32 state PE modes 13271

G1.10 AArch32 general-purpose registers, the PC, and the Special-purpose registers 13277

G1.11 Process state, PSTATE 13280

G1.12 Instruction set states 13286

G1.13 Handling exceptions that are taken to an Exception level using AArch32 13288

G1.14 Routing of aborts taken to AArch32 state 13306

G1.15 Exception return to an Exception level using AArch32 13308

G1.16 Asynchronous exception behavior for exceptions taken from AArch32 state 13313

G1.17 AArch32 state exception descriptions 13321

G1.18 Reset into AArch32 state 13343

G1.19 Mechanisms for entering a low-power state 13345

G1.20 The AArch32 System register interface 13350

G1.21 Advanced SIMD and floating-point support 13353

G1.22 Configurable instruction controls 13357

Chapter G2 **AArch32 Self-hosted Debug**

G2.1 About self-hosted debug 13363

G2.2 Routing debug exceptions 13367

G2.3 The debug exception enable controls 13369

G2.4 The effect of powerdown on debug exceptions 13371

G2.5 Summary of permitted routing and enabling of debug exceptions 13372

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.6 Pseudocode description of debug exceptions 13374
G2.7 Breakpoint Instruction exceptions 13375
G2.8 Breakpoint exceptions 13378
G2.9 Watchpoint exceptions 13402
G2.10 Vector Catch exceptions 13415
G2.11 Synchronization and debug exceptions 13423

Chapter G3 AArch32 Self-hosted Trace

G3.1 About self-hosted trace 13426
G3.2 Prohibited regions in self-hosted trace 13428
G3.3 Self-hosted trace timestamps 13429
G3.4 Synchronization in self-hosted trace 13430

Chapter G4 The AArch32 System Level Memory Model

G4.1 About the memory system architecture 13432
G4.2 Address space 13433
G4.3 Mixed-endian support in AArch32 13434
G4.4 AArch32 cache and branch predictor support 13435
G4.5 System register support for IMPLEMENTATION DEFINED memory features 13459
G4.6 External aborts 13460
G4.7 Memory barrier instructions 13462
G4.8 Pseudocode description of general memory System instructions 13463

Chapter G5 The AArch32 Virtual Memory System Architecture

G5.1 About VMSAv8-32 13467
G5.2 The effects of disabling address translation stages on VMSAv8-32 behavior 13475
G5.3 Translation tables 13479
G5.4 The VMSAv8-32 Short-descriptor translation table format 13484
G5.5 The VMSAv8-32 Long-descriptor translation table format 13493
G5.6 Memory access control 13512
G5.7 Memory region attributes 13523
G5.8 Translation Lookaside Buffers 13535
G5.9 TLB maintenance requirements 13539
G5.10 Caches in VMSAv8-32 13553
G5.11 VMSAv8-32 memory aborts 13556
G5.12 Exception reporting in a VMSAv8-32 implementation 13568
G5.13 Address translation instructions 13586
G5.14 Pseudocode description of VMSAv8-32 memory system operations 13592
G5.15 About the System registers for VMSAv8-32 13595

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.16 Functional grouping of VMSAv8-32 System registers 13600

Chapter G6 The Generic Timer in AArch32 state

G6.1 About the Generic Timer in AArch32 state 13602

G6.2 The AArch32 view of the Generic Timer 13607

Chapter G7 AArch32 System Register Encoding

G7.1 The AArch32 System register encoding space 13616

G7.2 Organization of registers in the (`coproc` == `0b1110`) encoding space 13617

G7.3 Organization of registers in the (`coproc` == `0b1111`) encoding space 13620

Chapter G8 AArch32 System Register Descriptions

G8.1 About the AArch32 System registers 13634

G8.2 General system control registers 13649

G8.3 Debug registers 14243

G8.4 Performance Monitors registers 14401

G8.5 Activity Monitors registers 14505

G8.6 RAS registers 14546

G8.7 Generic Timer registers 14623

Part H External Debug

Chapter H1 About External Debug

H1.1 Introduction to external debug 14720

H1.2 External debug 14722

H1.3 Required debug authentication 14723

Chapter H2 Debug State

H2.1 About Debug state 14726

H2.2 Halting the PE on debug events 14727

H2.3 Entering Debug state 14734

H2.4 Behavior in Debug state 14737

H2.5 Exiting Debug state 14771

Chapter H3 Halting Debug Events

H3.1 Introduction to Halting debug events 14774

H3.2 Halting Step debug events 14776

H3.3 Halt Instruction debug event 14786

H3.4 Exception Catch debug event 14787

H3.5 External Debug Request debug event 14791

H3.6 OS Unlock Catch debug event 14794

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.7 Reset Catch debug events 14795
H3.8 Software Access debug event 14796
H3.9 Synchronization and Halting debug events 14797

Chapter H4 The Debug Communication Channel and Instruction Transfer Register

H4.1 Introduction 14800
H4.2 DCC and ITR registers 14801
H4.3 DCC and ITR access modes 14805
H4.4 Flow control of the DCC and ITR registers 14809
H4.5 Synchronization of DCC and ITR accesses 14813
H4.6 Interrupt-driven use of the DCC 14818
H4.7 Pseudocode description of the operation of the DCC and ITR registers 14819

Chapter H5 The Embedded Cross-Trigger Interface

H5.1 About the Embedded Cross-Trigger 14821
H5.2 Basic operation on the ECT 14823
H5.3 Cross-triggers on a PE in an Arm A-profile implementation 14827
H5.4 Description and allocation of CTI triggers 14828
H5.5 CTI registers programmers' model 14832
H5.6 Examples 14833

Chapter H6 Debug Reset and Powerdown Support

H6.1 About Debug over powerdown 14837
H6.2 Power domains and debug 14838
H6.3 Core power domain power states 14839
H6.4 Powerup request mechanism 14841
H6.5 Emulating low-power states 14842
H6.6 Debug OS Save and Restore sequences 14844
H6.7 Reset and debug 14850

Chapter H7 The PC Sample-based Profiling Extension

H7.1 About the PC Sample-based Profiling Extension 14853

Chapter H8 About the External Debug Registers

H8.1 Relationship between external debug and System registers 14858
H8.2 Endianness and supported access sizes 14859
H8.3 Synchronization of changes to the external debug registers 14860
H8.4 Memory-mapped accesses to the external debug interface 14864
H8.5 External debug interface register access permissions 14866
H8.6 External debug interface registers 14870

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.7 Cross-trigger interface registers 14876
H8.8 External debug register resets 14878

Chapter H9 External Debug Register Descriptions

H9.1 About the external debug registers 14882
H9.2 External debug registers 14883
H9.3 External trace registers 15043
H9.4 External Trace Buffer registers 15235
H9.5 Cross-Trigger Interface registers 15302

Part I Memory-mapped Components of the Arm Architecture

Chapter I1 Requirements for Memory-mapped Components

I1.1 Supported access sizes 15365
I1.2 Synchronization of memory-mapped registers 15367
I1.3 Access requirements for reserved and unallocated registers 15369

Chapter I2 System Level Implementation of the Generic Timer

I2.1 About the Generic Timer specification 15371
I2.2 Memory-mapped counter module 15373
I2.3 Memory-mapped timer components 15377

Chapter I3 Recommended External Interface to the Performance Monitors

I3.1 About the external interface to the Performance Monitors registers 15382

Chapter I4 Recommended External Interface to the Activity Monitors

I4.1 About the external interface to the Activity Monitors Extension registers 15390

Chapter I5 External System Control Register Descriptions

I5.1 About the external system control register descriptions 15393
I5.2 External Performance Monitors registers summary 15395
I5.3 Performance Monitors external register descriptions 15400
I5.4 External Activity Monitors Extension registers summary 15566
I5.5 Activity Monitors external register descriptions 15569
I5.6 Generic Timer memory-mapped registers overview 15633
I5.7 Generic Timer memory-mapped register descriptions 15634

Part J Architectural Pseudocode

Chapter J1 A-profile Architecture Pseudocode

J1.1 Pseudocode for AArch64 operation 15687
J1.2 Pseudocode for AArch32 operation 16042

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3 Shared pseudocode 16148

Part K Appendixes

Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors

K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors 16455

K1.2 AArch64 CONSTRAINED UNPREDICTABLE behaviors 16476

Appendix K2 Recommendations for Reporting Memory Attributes on an Interconnect

K2.1 Arm recommendations for reporting memory attributes on an interconnect 16496

Appendix K3 ETE Recommended Configurations

K3.1 Configurations 16498

Appendix K4 Stages of Execution

K4.1 Stages of execution 16500

Appendix K5 Recommended External Debug Interface

K5.1 About the recommended external debug interface 16502

K5.2 PMUEVENT bus 16506

K5.3 Recommended authentication interface 16507

K5.4 Management registers and CoreSight compliance 16509

Appendix K6 Additional Information for Implementations of the Generic Timer

K6.1 Providing a complete set of features in a system level implementation 16521

K6.2 Gray count scheme for timer distribution scheme 16523

Appendix K7 Legacy Instruction Syntax for AArch32 Instruction Sets

K7.1 Legacy Instruction Syntax 16525

Appendix K8 Address Translation Examples

K8.1 AArch64 Address translation examples 16532

K8.2 AArch32 Address translation examples 16546

Appendix K9 Recommended Upload and Download Processes for External Debug

K9.1 Using Memory access mode in AArch64 state 16556

Appendix K10 Software Usage Examples

K10.1 Use of the Advanced SIMD complex number instructions 16561

K10.2 Use of the Armv8.2 extensions to the Cryptographic Extension 16563

K10.3 SVE and SVE2 Programming Examples 16570

K10.4 Use of the FEAT_PCDPHINT STSHH and PRFM IR (immediate) hint instructions 16571

K10.5 Use of the Embedded Trace and Trace Buffer Extensions 16573

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.6 Use of the GCS Extension 16638
K10.7 Use of the RAS Extension 16648
K10.8 Example OS Save and Restore sequences 16649

Appendix K11 Barrier Litmus Tests

K11.1 Introduction 16653
K11.2 Load-Acquire, Store-Release and barriers 16656
K11.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers 16659
K11.4 Using a mailbox to send an interrupt 16664
K11.5 Cache and TLB maintenance instructions and barriers 16665

Appendix K12 Random Number Generation

K12.1 Properties of the generated random number 16678

Appendix K13 Arm Pseudocode Definition

K13.1 About the Arm pseudocode 16680
K13.2 Pseudocode for instruction descriptions 16681
K13.3 Data types 16683
K13.4 Operators 16688
K13.5 Statements and control structures 16694
K13.6 Built-in functions 16699
K13.7 Miscellaneous helper procedures and functions 16702
K13.8 Arm pseudocode definition index 16704

Appendix K14 Registers Index

K14.1 Introduction and register disambiguation 16708
K14.2 Alphabetical index of AArch64 registers and System instructions 16719
K14.3 Functional index of AArch64 registers and System instructions 16739
K14.4 Alphabetical index of AArch32 registers and System instructions 16759
K14.5 Functional index of AArch32 registers and System instructions 16768
K14.6 Alphabetical index of memory-mapped registers 16778
K14.7 Functional index of memory-mapped registers 16791

Glossary

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

This preface introduces the *Arm Architecture Reference Manual, for A-profile architecture*. It contains the following sections:

- About this Manual.
- Using this Manual.
- Conventions.
- Additional reading.
- Feedback.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this Manual

This Manual describes the Arm® architecture v8, Armv8, and the Arm® architecture v9, Armv9. The architecture describes the operation of an Armv8-A and an Armv9-A Processing element (PE), and this Manual includes descriptions of:

- The two Execution states, AArch64 and AArch32.
- The instruction sets:
  - In AArch32 state, the T32 and A32 instruction sets, which are compatible with earlier versions of the Arm architecture.
  - In AArch64 state, the A64 instruction set.
- The states that determine how a PE operates, including the current Exception level and Security state, and in AArch32 state the PE mode.
- The Exception model.
- The interprocessing model, that supports transitioning between AArch64 state and AArch32 state.
- The memory model, that defines memory ordering and memory management. This Manual covers the Arm A architecture profile, both Armv8-A and Armv9-A, that defines a *Virtual Memory System Architecture* (VMSA).
- The programmers' model, and its interfaces to System registers that control most PE and memory system features, and provide status information.
- The Advanced SIMD and floating-point instructions, which provide high-performance:
  - Single-precision, half-precision, and double-precision floating-point operations.
  - Conversions between double-precision, single-precision, and half-precision floating-point values.
  - Integer, single-precision floating-point, and half-precision floating-point vector operations in all instruction sets.
  - Double-precision floating-point vector operations in the A64 instruction set.
- The security model, which provides up to four Security states to support Secure applications and confidential computing.
- The virtualization model.
- The Debug architecture, which provides software access to debug features.

This Manual gives the assembler syntax for the instructions it describes, meaning that it describes instructions in textual form. However, this Manual is not a tutorial for Arm assembler language, nor does it describe Arm assembler language, except at a basic level. To make effective use of Arm assembler language, read the documentation supplied with the assembler being used.

This Manual is organized into parts:

Part A Provides an introduction to the Arm architecture, and an overview of the AArch64 and AArch32 Execution states.

Part B Describes the application level view of the AArch64 Execution state, meaning the view from EL0. It describes the application level view of the programmers' model and the memory model.

Part C Describes the A64 instruction set, which is available in the AArch64 Execution state. The descriptions for each instruction also include the precise effects of each instruction when executed at EL0, described as *unprivileged* execution, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Part D Describes the system level view of the AArch64 Execution state. It includes details of the System registers, most of which are not accessible from EL0, and the system level view of the programmers' model and the memory model. This part includes the description of self-hosted debug.

Part E Describes the application level view of the AArch32 Execution state, meaning the view from the EL0. It describes the application level view of the programmers' model and the memory model.

Note

In AArch32 state, execution at EL0 is execution in User mode.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part F Describes the T32 and A32 instruction sets, which are available in the AArch32 Execution state. These instruction sets are backwards-compatible with earlier versions of the Arm architecture. This part describes the precise effects of each instruction when executed in User mode, described as *unprivileged* execution or execution at EL0, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Note

User mode is the only mode where software execution is unprivileged.

Part G Describes the system level view of the AArch32 Execution state, which is generally compatible with earlier versions of the Arm architecture. This part includes details of the System registers, most of which are not accessible from EL0, and the instruction interface to those registers. It also describes the system level view of the programmers' model and the memory model.

Part H Describes the Debug architecture for external debug. This provides configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

Part I Describes additional features of the architecture that are not closely coupled to a PE, and therefore are accessed through memory-mapped interfaces. Some of these features are OPTIONAL.

Part J Provides pseudocode that describes various features of the Arm architecture.

Part K, Appendixes

Provide additional information. Some appendixes give information that is not part of the A-profile architectural requirements. The cover page of each appendix indicates its status.

Glossary Defines terms used in this Manual that have a specialized meaning.

Note

Terms that are generally well understood in the microelectronics industry are not included in the Glossary.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Using this Manual

The information in this Manual is organized into parts, as described in this section.

Part A, Introduction and Architecture Overview

Part A gives an overview of the Armv8-A and Armv9-A architecture profiles, including its relationship to the other Arm PE architectures. It introduces the terminology used to describe the architecture, and gives an overview of the Executions states, AArch64 and AArch32.

It contains the following chapters:

Introduction to the Arm Architecture

Read this for an introduction to the Arm architecture.

A-profile Architecture Extensions

Read this for an introduction to the A-profile architecture extensions and features.

Part B, The AArch64 Application Level Architecture

Part B describes the AArch64 state application level view of the architecture. It contains the following chapters:

The AArch64 Application Level Programmers' Model

Read this for an application level description of the programmers' model for software executing in AArch64 state. It describes execution at EL0 when EL0 is using AArch64 state.

The AArch64 Application Level Memory Model

Read this for an application level description of the memory model for software executing in AArch64 state. It describes the memory model for execution in EL0 when EL0 is using AArch64 state. It includes information about Arm memory types, attributes, and memory access controls.

Part C, The A64 Instruction Set

Part C describes the A64 instruction set, which is used in AArch64 state. It contains the following chapters:

The A64 Instruction Set

Read this for a description of the A64 instruction set and common instruction operation details.

About the A64 Instruction Descriptions

Read this to understand the format of the A64 instruction descriptions.

A64 Instruction Set Overview

Read this for an overview of the A64 instructions.

A64 Instruction Set Encoding

Read this for a description of the A64 instruction set encoding.

The A64 System Instruction Class

Read this for a description of the AArch64 System instructions and register descriptions, and the System instruction class encoding space.

A64 Base Instruction Descriptions

Read this for information on key aspects of the A64 base instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A64 Advanced SIMD and Floating-point Instruction Descriptions

Read this for information on key aspects of the A64 Advanced SIMD and floating-point instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

SVE Instruction Descriptions

Read this for information on key aspects of the SVE instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

SME Instruction Descriptions

Read this for information on key aspects of the SME instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Part D, The AArch64 System Level Architecture

Part D describes the AArch64 state system level view of the architecture. It contains the following chapters:

The AArch64 System Level Programmers' Model

Read this for a description of the AArch64 state system level view of the programmers' model.

AArch64 Self-hosted Debug

Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

AArch64 Self-hosted Trace

Read this for an introduction to, and a description of, self-hosted trace in AArch64 state.

The Embedded Trace Extension

Read this for a description of the Embedded Trace Extension.

ETE Protocol Descriptions

Read this for a description of the ETE packets, which are listed in alphabetical order.

The Trace Buffer Extension

Read this for a description of the Trace Buffer Extension.

The AArch64 System Level Memory Model

Read this for a description of the AArch64 state system level view of the general features of the memory system.

The AArch64 Virtual Memory System Architecture

Read this for a system level view of the AArch64 *Virtual Memory System Architecture* (VMSA), the memory system architecture of an A-profile architecture implementation executing in AArch64 state.

The Granule Protection Check Mechanism

Read this for a description of the granule protection check mechanism, a feature that is added as part of the OPTIONAL Realm Management Extension.

The Memory Tagging Extension

Read this for a description of the Memory Tagging Extension.

The Guarded Control Stack

Read this for a description of the Guarded Control Stack extension.

The Generic Timer in AArch64 state

Read this for a description of the AArch64 view of the Arm Generic Timer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Performance Monitors Extension

Read this for a description of the Arm Performance Monitors, an optional non-invasive debug component.

The System Performance Monitors Extension

Read this for a description of the System Performance Monitors, an optional non-invasive debug component.

The Activity Monitors Extension

Read this for a description of the Arm Activity Monitors.

The Statistical Profiling Extension

Read this for a description of the Statistical Profiling Extension, an optional AArch64 state non-invasive debug component.

Statistical Profiling Extension Sample Record Specification

Read this for a description of the sample records generated by the Statistical Profiling Extension.

The Branch Record Buffer Extension

Read this for a description of the Branch Record Buffer Extension.

RAS PE Architecture

Read this for a description of the RAS Extension PE architecture.

MPAM PE Architecture

Read this for a description of the MPAM Extension PE architecture.

The Scalable Matrix Extension

Read this for a description of the Scalable Matrix Extension.

AArch64 System Register Encoding

Read this for a description of the encoding of the AArch64 System registers, and the other uses of the AArch64 System registers encoding space.

AArch64 System Register Descriptions

Read this for an introduction to, and description of, each of the AArch64 System registers.

Part E, The AArch32 Application Level Architecture

Part E describes the AArch32 state application level view of the architecture. It contains the following chapters:

The AArch32 Application Level Programmers' Model

Read this for an application level description of the programmers' model for software executing in AArch32 state. It describes execution at EL0 when EL0 is using AArch32 state.

The AArch32 Application Level Memory Model

Read this for an application level description of the memory model for software executing in AArch32 state. It describes the memory model for execution in EL0 when EL0 is using AArch32 state. It includes information about Arm memory types, attributes, and memory access controls.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part F, The AArch32 Instruction Sets

Part F describes the T32 and A32 instruction sets, which are used in AArch32 state. It contains the following chapters:

About the T32 and A32 Instruction Descriptions

Read this to understand the format of the T32 and A32 instruction descriptions.

The AArch32 Instruction Sets Overview

Read this for an overview of the T32 and A32 instruction sets.

T32 Instruction Set Encoding

Read this for a description of the T32 instruction set encoding. This includes the T32 encoding of the Advanced SIMD and floating-point instructions.

A32 Instruction Set Encoding

Read this for a description of the A32 instruction set encoding. This includes the A32 encoding of the Advanced SIMD and floating-point instructions.

T32 and A32 Base Instruction Set Instruction Descriptions

Read this for a description of each of the T32 and A32 base instructions.

T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

Read this for a description of each of the T32 and A32 Advanced SIMD and floating-point instructions.

Part G, The AArch32 System Level Architecture

Part G describes the AArch32 state system level view of the architecture. It contains the following chapters:

The AArch32 System Level Programmers' Model

Read this for a description of the AArch32 state system level view of the programmers' model for execution in an Exception level that is using AArch32.

AArch32 Self-hosted Debug

Read this for an introduction to, and a description of, self-hosted debug in AArch32 state.

AArch32 Self-hosted Trace

Read this for an introduction to, and a description of, self-hosted trace in AArch32 state.

The AArch32 System Level Memory Model

Read this for a system level view of the general features of the memory system.

The AArch32 Virtual Memory System Architecture

Read this for a description of the AArch32 Virtual Memory System Architecture (VMSA).

The Generic Timer in AArch32 state

Read this for a description of the AArch32 view of an implementation of the Arm Generic Timer.

AArch32 System Register Encoding

Read this for a description of the encoding of the AArch32 System registers, including the System instructions that are part of the AArch32 System registers encoding space.

AArch32 System Register Descriptions

Read this for a description of each of the AArch32 System registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part H, External Debug

Part H describes the architecture for external debug. It contains the following chapters:

About External Debug

Read this for an introduction to external debug, and a definition of the scope of this part of the Manual.

Debug State

Read this for a description of Debug state, which the PE might enter as the result of a Halting debug event.

Halting Debug Events

Read this for a description of the external debug events referred to as Halting debug events.

The Debug Communication Channel and Instruction Transfer Register

Read this for a description of the communication between a debugger and the PE debug logic using the Debug Communications Channel and the Instruction Transfer register.

The Embedded Cross-Trigger Interface

Read this for a description of the Embedded Cross-Trigger Interface.

Debug Reset and Powerdown Support

Read this for a description of reset and powerdown support in the Debug architecture.

The PC Sample-based Profiling Extension

Read this for a description of the PC Sample-based Profiling Extension that is an OPTIONAL extension to an Armv8 or Armv9 implementation.

About the External Debug Registers

Read this for some additional information about the external debug registers.

External Debug Register Descriptions

Read this for a description of each external debug register.

Part I, Memory-mapped Components of the Arm architecture

Part I describes the memory-mapped components in the architecture. It contains the following chapters:

Requirements for Memory-mapped Components

Read this for descriptions of some general requirements for memory-mapped components within a system that complies with the Arm architecture.

System Level Implementation of the Generic Timer

Read this for a definition of a system level implementation of the Generic Timer.

Recommended External Interface to the Performance Monitors

Read this for a description of the recommended memory-mapped and external debug interfaces to the Performance Monitors.

Recommended External Interface to the Activity Monitors

Read this for a description of the recommended memory-mapped interface to the Activity Monitors.

External System Control Register Descriptions

Read this for a description of each memory-mapped system control register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part J, Architectural Pseudocode

Part J contains pseudocode that describes various features of the Arm architecture. It contains the following chapter:

A-profile Architecture Pseudocode

Read this for the pseudocode definitions that describe various features of the Arm architecture, for operation in AArch64 state and in AArch32 state.

Part K, Appendixes

Note

Some of the descriptions in the following appendixes are not part of the Arm architecture specification. They are included here as supplementary information, for the convenience of developers and users who might require this information.

This Manual contains the following appendixes:

Architectural Constraints on UNPREDICTABLE Behaviors

Read this for a description of the architecturally-required constraints on UNPREDICTABLE behaviors in the Arm architecture, including AArch32 behaviors that were UNPREDICTABLE in previous versions of the architecture.

Recommendations for Reporting Memory Attributes on an Interconnect

Read this for the Arm recommendations about how the architectural memory attributes are reported on an interconnect.

ETE Recommended Configurations

Read this for a description of the ETE recommended configurations.

Stages of Execution

Read this for a description of the stages of execution.

Recommended External Debug Interface

Read this for a description of the recommended external debug interface.

Additional Information for Implementations of the Generic Timer

Read this for additional information about implementations of the Arm Generic Timer. This information does not form part of the architectural definition of the Generic Timer.

Legacy Instruction Syntax for AArch32 Instruction Sets

Read this for information about the pre-UAL syntax of the AArch32 instruction sets, which are valid for the A32 instruction set.

Address Translation Examples

Read this for examples of translation table lookups using the translation regimes described in The AArch64 Virtual Memory System Architecture and The AArch32 Virtual Memory System Architecture.

Recommended Upload and Download Processes for External Debug

Read this for information about implementing and using the Arm architecture.

Software Usage Examples

Read this for software examples that help understanding of some aspects of the Arm architecture.

Barrier Litmus Tests

Read this for examples of the use of barrier instructions provided by the Arm architecture.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Random Number Generation

Read this for information on the generation of random numbers using FEAT_RNG.

Arm Pseudocode Definition

Read this for definitions of the Arm pseudocode.

Registers Index

Read this for an alphabetic and functional index of AArch32 and AArch64 registers, and memory-mapped registers.

Glossary

Defines terms used in this Manual that have a specialized meaning.

Note

Terms that are well understood in the microelectronics industry are not included in the Glossary.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Conventions

The following sections describe conventions that this book can use:

- Typographic conventions.
- Rules-based writing.
- Signals.
- Numbers.
- Pseudocode descriptions.
- Assembler syntax descriptions.

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

bold Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, and are defined in the Glossary.

Colored text Indicates a link. This can be:

- A URL, for example https://developer.arm.com.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the Manual that defines the colored term, for example Simple sequential execution or SCTLR.

{ and } Braces, { and }, have two distinct uses:

Optional items

In syntax descriptions braces enclose optional items. In the following example they indicate that the `<shift>`parameter is optional:

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

Similarly, they can be used in generalized field descriptions, for example TCR_ELx.{I}PS refers to a field in the TCR_ELx registers that is called either IPS or PS.

Sets of items

Braces can be used to enclose sets. For example, HCR_EL2.{E2H, TGE} refers to a set of two register fields, HCR_EL2.E2H and HCR_EL2.TGE.

Notes Notes are formatted as:

Note

This is a Note.

In this Manual, Notes are used only to provide additional information, usually to help understanding of the text. While a Note might repeat architectural information given elsewhere in the Manual, a Note never provides any part of the definition of the architecture.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rules-based writing

Some sections of this Manual use rules-based writing. Rules-based writing consists of a set of individual content items. A content item is classified as one of the following:

- Rule.
- Information.
- Software usage.
- Declaration.

Rules are normative statements. An implementation that is compliant with this specification must conform to all Rules in this Manual that apply to that implementation.

Rules must not be read in isolation. Where a particular feature is specified by multiple Rules, these are generally grouped into sections and subsections that provide context. Where appropriate, these sections begin with a short introduction.

Arm strongly recommends that implementers read all chapters and sections of this Manual to ensure that an implementation is compliant.

Content items other than Rules are informative statements. These are provided as an aid to understanding this Manual.

Content item identifiers

A content item may have an associated identifier which is unique among content items in this Manual. After content reaches beta status, a given content item has the same identifier across subsequent versions of this Manual.

Content item rendering

In this Manual, a content item is rendered with a token of the following format in the left margin: L_{iiiii}. L is a label that indicates the content class of the content item. iiiii is the identifier of the content item.

Content item classes

Each of the content item classes has a different function in this Manual.

Rule

A Rule is a statement that describes the behavior of a compliant implementation.

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label R.

Information

An Information statement provides information and guidance as an aid to understanding the Manual.

An Information statement is rendered with the label I.

Software usage

A Software usage statement provides guidance on how software can make use of the features defined by the specification.

A Software usage statement is rendered with the label S.

Declaration

A Declaration statement introduces concepts or terminology.

A Declaration does not describe behavior.

A Declaration is rendered with the label D.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signals

In general this specification does not define hardware signals, but it does include some signal examples and recommendations. The signal conventions are:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n At the start or end of a signal name denotes an active-LOW signal.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF_0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This Manual uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in `monospace` font, and is described in Arm Pseudocode Definition.

Assembler syntax descriptions

This Manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font, and use the conventions described in Structure of the A64 assembler language, and Arm Pseudocode Definition.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additional reading

This section lists relevant publications from Arm and third parties.

See Arm Developer, https://developer.arm.com, for access to Arm documentation.

Arm publications

- *AMBA® 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1* (ARM IHI 0032).
- *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification* (ARM IHI 0099).
- *Arm® Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile* (ARM DDI 0600).
- *ARM® Architecture Reference Manual Supplement - Armv8, for the ARMv8-R AArch32 architecture profile* (ARM DDI 0568).
- *Arm® Architecture Reference Manual Supplement, Transactional Memory Extension (TME), for A-profile architecture* (ARM DDI 0617).
- *Arm® Debug Interface Architecture Specification, ADIv6.0* (ARM IHI 0074).
- *Arm® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
- *Arm® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).
- *ARM® CoreSight™ SoC Technical Reference Manual* (ARM DDI 0480).
- *Arm® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *ARM® Procedure Call Standard for the ARM 64-bit Architecture* (ARM IHI 0055).
- *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).
- *Arm® Server Base System Architecture* (ARM DEN 0029).
- *Arm® Base System Architecture* (ARM DEN 0094).

Other publications

The following publications are referred to in this Manual, or provide more information:

- *Announcing the Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, November 2001.
- IEEE Std 754-2008, *IEEE Standard for Floating-point Arithmetic*, August 2008.
- IEEE Std 754-1985, *IEEE Standard for Floating-point Arithmetic*, March 1985.
- *Secure Hash Standard (SHA)*, Federal Information Processing Standards Publication 180-2, August 2002.
- *The Galois/Counter Mode of Operation*, McGraw, D. and Viega, J., Submission to NIST Modes of Operation Process, January 2004.
- *Memory Consistency Models for Shared Memory-Multiprocessors*, Gharachorloo, Kourosh, 1995, Stanford University Technical Report CSL-TR-95-685.
- *Standard Manufacturer's Identification Code, JEP106*, JEDEC Solid State Technology Association.
- *SM3 Cryptographic Hash Algorithm*, China Internet Network Information Center (CNNIC).
- *SM4 Block Cipher Algorithm*, China Internet Network Information Center (CNNIC).
- *The QARMA Block Cipher Family*, Roberto Avanzi, Qualcomm Product Security Initiative. Available from https://eprint.iacr.org/2016/444.
- *OCP 8-bit Floating Point Specification (OFP8)* Paulius Micikevicius, Stuart Oberman, Pradeep Dubey, Marius Cornea, Andres Rodriguez, Ian Bratt, Richard Grisenthwaite, et al., June 2023. Available from https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Feedback

Arm welcomes feedback on its documentation.

Feedback on this Manual

If you have any comments or queries about this Manual, create a ticket at https://support.developer.arm.com.

As part of the ticket, include:

- The title, Arm® Architecture Reference Manual, for A-profile architecture.
- The number, ARM DDI 0487 M.a.a.
- The section name to which your comments refer.
- The page number(s) to which your comments refer.
- The rule identifier(s) to which your comments refer, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader. Search performance may be significantly improved by increasing the Fast Find Maximum Cache Size under Search in Preferences.
If using Google Chrome™ or Microsoft Edge™, load time is significantly improved since the April 2025 release of these browsers.
Users may find, on some systems, that PDF viewers such as Okular or Foxit™ give good performance.

Inclusive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

Previous issues of this document included language that can be offensive. We have replaced this language. To report offensive language in this document, email `terms@arm.com`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part A
Arm Architecture Introduction and Overview

Chapter A1
Introduction to the Arm Architecture

This chapter introduces the Arm architecture. It contains the following sections:

- About the Arm architecture.
- Architecture profiles.
- Arm architectural concepts.
- Supported data types.
- Floating-point support.
- The Arm memory model.
- Reliability, Availability, and Serviceability.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.1 About the Arm architecture

The Arm architecture described in this Architecture Reference Manual defines the behavior of an abstract machine, referred to as a *processing element*, often abbreviated to *PE*. Implementations compliant with the Arm architecture must conform to the described behavior of the processing element. It is not intended to describe how to build an implementation of the PE, nor to limit the scope of such implementations beyond the defined behaviors.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with the Arm architecture must be the same as a simple sequential execution of the program on the processing element. The programmer-visible behavior does not include a specified execution time for any instruction but requires each instruction to execute in finite time, as long as another observer is not continually modifying any Location associated with a Memory Effect, other than an Explicit Memory Effect, that is architecturally required for execution of the instruction. In cases where another observer is continually modifying a Location associated with a Memory Effect, other than an Explicit Memory Effect, at least one of the observers must make forward progress in finite time. Unless otherwise specified, the architecturally defined effects of each instruction also complete in finite time.

The Arm Architecture Reference Manual also describes rules for software to use the PE.

The Arm architecture includes definitions of:

- An associated debug architecture, see:
  - AArch64 Self-hosted Debug.
  - AArch32 Self-hosted Debug.
  - External Debug.
- Associated trace architectures that define trace units that implementers can implement with the associated processor hardware. For more information, see:
  - The *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
  - AArch64 Self-hosted Trace.
  - The Embedded Trace Extension.
  - The Trace Buffer Extension.
  - AArch32 Self-hosted Trace.

Arm also provides system architecture specifications for systems aimed at particular product categories, such as servers, or personal computers, *Arm® Server Base System Architecture* (ARM DEN 0029) and *Arm® Base System Architecture* (ARM DEN 0094). These specifications build on the architecture specification in this Manual, restricting, as defined in those specifications, some of the optional and IMPLEMENTATION DEFINED features defined in this specification, as well as specifying additional features applicable at the system level.

The Arm architecture is a *Reduced Instruction Set Computer* (RISC) architecture with the following RISC architecture features:

- A large uniform register file.
- A *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system.

This document defines versions 8 and 9 of the A-profile architecture.

See Architecture profiles for more information.

The Arm architecture supports implementations across a wide range of performance points. Implementation size, performance, and very low power consumption are key attributes of the Arm architecture.

An important feature of the Arm architecture is backward compatibility, combined with the freedom for optimal implementation in a wide range of standard and more specialized use cases.

The Arm architecture supports:

- A 64-bit Execution state, AArch64.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A 32-bit Execution state, AArch32, that is compatible with previous versions of the Arm architecture.

Features that are optional are explicitly defined as such in this Manual.

Note

The presence of an ID register field for a feature does not imply that the feature is optional.

Both Execution states support floating-point instructions:

- AArch32 state provides:
  - SIMD instructions in the base instruction sets that operate on the 32-bit general-purpose registers.
  - Advanced SIMD instructions that operate on registers in the *SIMD and floating-point register* (SIMD&FP register) file.
  - Scalar floating-point instructions that operate on registers in the SIMD&FP register file.
- AArch64 state provides:
  - Advanced SIMD instructions that operate on registers in the SIMD&FP register file.
  - Scalar floating-point instructions that operate on registers in the SIMD&FP register file.

    Note

    The A64 instruction set does not include SIMD instructions that operate on the general-purpose registers, therefore, some AArch64 instructions descriptions use SIMD as a synonym for Advanced SIMD.

  - SVE instructions that operate on registers in the SVE register files.
  - SME instructions that operate on registers in the SVE register files and ZA storage.

Note

See Conventions for information about conventions used in this Manual, including the use of SMALL CAPITALS for particular terms that have Arm-specific meanings that are defined in the Glossary.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.2 Architecture profiles

The Arm architecture has evolved significantly since its introduction, and Arm continues to develop it. Nine major versions of the architecture have been defined to date, denoted by the version numbers 1 to 9. Of these, the first three versions are now obsolete.

The generic names AArch64 and AArch32 describe the 64-bit and 32-bit Execution states:

AArch64 Is the 64-bit Execution state, meaning addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

AArch32 Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

Note

The *Base instruction set* comprises the supported instructions other than the floating-point instructions.

See sections Execution state and The instruction sets for more information.

Arm defines three architecture profiles:

A Application profile, described in this Manual:

- Supports a *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).

  Note

  An Armv8-A implementation can be called an AArchv8-A implementation and an Armv9-A implementation can be called an AArchv9-A implementation.

- Supports the A64, A32, and T32 instruction sets.

R Real-time profile:

- Supports a *Protected Memory System Architecture* (PMSA) based on a *Memory Protection Unit* (MPU).
- Supports an optional VMSA based on an MMU.
- Supports the A64, or the A32 and T32 instruction sets.

M Microcontroller profile:

- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
- Supports a PMSA based on an MPU.
- Supports a variant of the T32 instruction set.

This Manual describes only Armv8-A and Armv9-A. For information about the R and M architecture profiles, and earlier Arm architecture versions, see:

- The *Arm® Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile*.
- The *ARM® Architecture Reference Manual Supplement - ARMv8, for the ARMv8-R AArch32 architecture profile*.
- The *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition*.
- The *Armv8-M Architecture Reference Manual*.
- The *Armv7-M Architecture Reference Manual*.
- The *ARMv6-M Architecture Reference Manual*.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.3 Arm architectural concepts

This section introduces architectural concepts and the associated terminology.

The following subsections describe key architectural concepts. Each section introduces the corresponding terms that are used to describe the architecture:

- Execution state.
- The instruction sets.
- System registers.
- Arm Debug.

A1.3.1 Execution state

The Execution state defines the PE execution environment, including:

- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
  - The Exception model.
  - The *Virtual Memory System Architecture* (VMSA).
  - The programmers' model.

The Execution states are:

AArch64 The 64-bit Execution state. This Execution state:

- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit *Program Counter* (PC), *stack pointers* (SPs), and *Exception Link Registers* (ELRs).
- Provides 32 128-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64. For more information, see The instruction sets.
- Defines the Armv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an *execution privilege* hierarchy, see Exception levels.
- Provides support for 64-bit *virtual addressing*. For more information, including the limits on address ranges, see The AArch64 Virtual Memory System Architecture.
- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
- Names each System register using a suffix that indicates the lowest Exception level at which the register can be accessed.

AArch32 The 32-bit Execution state. This Execution state:

- Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and *Link Register* (LR). The LR is used as both an ELR and a procedure link register.
  Some of these registers have multiple *banked* instances for use in different PE *modes*.
- Provides a single ELR, for exception returns from Hyp mode.
- Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides two instruction sets, T32 and A32. For more information, see The instruction sets.
- Supports the Armv7-A Exception model, based on *PE modes*, and maps this onto the Armv8 Exception model, that is based on the Exception levels.
- Provides support for 32-bit virtual addressing.
- Defines a number of *Process state* (PSTATE) elements that hold PE state. The T32 and A32 instruction sets include instructions that operate directly on various PSTATE elements, and instructions that access PSTATE by using the *Application Program Status Register* (APSR) or the *Current Program Status Register* (CPSR).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Later subsections give more information about the different properties of the Execution states.

Transferring control between the AArch64 and AArch32 Execution states is known as *interprocessing*. The PE can move between Execution states only on a change of Exception level, and subject to the rules given in Interprocessing. This means different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.

A1.3.2 The instruction sets

The possible instruction sets depend on the Execution state:

AArch64 AArch64 state supports a single instruction set, called A64. This is a fixed-length instruction set that uses 32-bit instruction encodings.

For information on the A64 instruction set, see A64 Instruction Set Overview.

If FEAT_SVE is implemented, the A64 instruction set supports scalable vector instructions. See About the SVE instructions.

If FEAT_SME is implemented, the A64 instruction set supports scalable matrix instructions. See About the SME instructions.

AArch32 AArch32 state supports the following instruction sets:

A32 This is a fixed-length instruction set that uses 32-bit instruction encodings.

T32 This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings.

In previous documentation, these instruction sets were called the ARM and Thumb® instruction sets. Armv8 and Armv9 extend each of these instruction sets. In AArch32 state, the Instruction set state determines the instruction set that the PE executes.

For information on the T32 and A32 instruction sets, see The AArch32 Instruction Sets Overview.

The instruction sets support SIMD and scalar floating-point instructions. See Floating-point support.

A1.3.3 System registers

System registers provide control and status information of architected features.

The System registers use a standard naming format: <register_name>.<bit_field_name> to identify specific registers as well as control and status bits within a register.

Bits can also be described by their numerical position in the form <register_name>[x:y] or the generic form bits[x:y].

In addition, in AArch64 state, most register names include the lowest Exception level that can access the register as a suffix to the register name:

- <register_name>_ELx, where x is 0, 1, 2, or 3.

For information about Exception levels, see Exception levels.

The System registers comprise:

- The following registers that are described in this Manual:
  - General system control registers.
  - Debug registers.
  - RAS registers.
  - Generic Timer registers.
  - Optionally, Performance Monitor registers.
  - Optionally, the Activity Monitors registers.
  - Optionally, the Scalable Vector Extension registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Optionally, in Armv9, Trace System registers.
- Optionally, one or more of the following groups of registers that are defined in other Arm architecture specifications:
  - Trace System registers, as defined in the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4*.
  - *Generic Interrupt Controller* (GIC) System registers, see The Arm Generic Interrupt Controller System registers.

For information about the AArch64 System registers, see AArch64 System Register Descriptions.

For information about the AArch32 System registers, see AArch32 System Register Descriptions.

A1.3.3.1 The Arm Generic Interrupt Controller System registers

From version 3 of the Arm Generic Interrupt Controller architecture, GICv3, the GIC architecture specification defines a System register interface to some of its functionality. The System register summaries in this Manual include these registers, see:

- About the GIC System registers, for more information about the AArch64 GIC System registers.
- About the GIC System registers, for more information about the AArch32 GIC System registers.

These sections give only short overviews of the GIC System registers. For more information, including descriptions of the registers, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).

Note

The programmers' model for earlier versions of the GIC architecture is wholly memory-mapped.

A1.3.4 Arm Debug

Armv8 and later architectures support the following:

Self-hosted debug

In this model, the PE generates *debug exceptions*. Debug exceptions are part of the Armv8 Exception model.

External debug

In this model, *debug events* cause the PE to enter *Debug state*. In Debug state, the PE is controlled by an external debugger.

All Armv8 and later implementations support both models. The model chosen by a particular user depends on the debug requirements during different stages of the design and development life cycle of the product. For example, external debug might be used during debugging of the hardware implementation and OS bring-up, and self-hosted debug might be used during application development.

For more information about self-hosted debug:

- In AArch64 state, see AArch64 Self-hosted Debug.
- In AArch32 state, see AArch32 Self-hosted Debug.

For more information about external debug, see External Debug.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.4 Supported data types

The Arm architecture supports the following integer data types:

Byte 8 bits.

Halfword 16 bits.

Word 32 bits.

Doubleword 64 bits.

Quadword 128 bits.

The architecture also supports the following floating-point data types:

- Half-precision. See Half-precision floating-point formats.
- Single-precision. See Single-precision floating-point format.
- Double-precision. See Double-precision floating-point format.
- BFloat16. See BFloat16 floating-point format.
- 8-bit floating-point. See FP8 floating-point formats.

It also supports vectors, where a register holds a vector of multiple elements, each the same size and data type. See Advanced SIMD vector formats and SVE vector formats.

The architecture provides the following:

- A general-purpose register file.
- A SIMD&FP register file.
- If FEAT_SVE or FEAT_SME is implemented, an SVE scalable vector register file and an SVE scalable predicate register file.
- If FEAT_SME is implemented, the scalable ZA storage.
- If FEAT_SME2 is implemented, the SME2 ZT0 register.

In each of these, the possible register widths depend on the Execution state.

In AArch64 state:

- The general-purpose register file contains 64-bit registers:
  - Many instructions can access these registers as 64-bit registers, or as 32-bit registers using only the bottom 32 bits.
- The SIMD&FP register file contains 128-bit registers:
  - While the AArch64 vector registers support 128-bit vectors, the effective vector length can be 64-bits or 128-bits depending on the A64 instruction encoding used, see Instruction Mnemonics.
- The SVE scalable vector register file contains registers of an IMPLEMENTATION DEFINED width:
  - An SVE scalable vector register has an IMPLEMENTATION DEFINED width that is a power of two, from a minimum of 128 bits up to a maximum of 2048 bits.
  - All SVE scalable vector registers in an implementation are the same width.
- The SVE scalable predicate register file contains registers of an IMPLEMENTATION DEFINED width:
  - An SVE predicate register has an IMPLEMENTATION DEFINED width that is a power of two, from a minimum of 16 bits up to a maximum of 256 bits.
- The SME scalable ZA storage is a two-dimensional array of bytes. See ZA storage.
- The SME2 ZT0 register is 512 bits. See SME2 ZT0 register.

For more information on the register files in AArch64 state, see Registers in AArch64 Execution state.

In AArch32 state:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The general-purpose register file contains 32-bit registers:
  - Two 32-bit registers can support a doubleword.
  - Vector formatting is supported, see Figure A1-4.
- The SIMD&FP register file contains 64-bit registers:
  - AArch32 state does not support quadword integer or floating-point data types.

Note

Two consecutive 64-bit registers can be used as a 128-bit register.

For more information on the register files in AArch32 state, see The general-purpose registers, and the PC, in AArch32 state.

A1.4.1 Advanced SIMD vector formats

In an implementation that includes the Advanced SIMD instructions that operate on the SIMD&FP register file, a register can hold one or more packed elements, all of the same size and type. In AArch32 state, the combination of a register and a data type describes a vector of elements, where the number of elements in the vector is implied by the size of the data type and the size of the register. In AArch64 state, the explicit combination of a register, number of elements, and element size describes a vector of elements. The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction.

Vector indices are in the range 0 to (number of elements - 1). An index of 0 refers to the least significant bits of the vector.

For more information on the Advanced SIMD and floating-point registers in AArch32 state, see The SIMD and floating-point register file.

A1.4.1.1 Advanced SIMD vector formats in AArch64 state

In AArch64 state, the SIMD&FP registers can be referred to as V*n*, where *n* is a value from 0 to 31.

The SIMD&FP registers support the following data formats for loads, stores, and data-processing operations:

- A single, scalar, element in the least significant bits of the register. See Registers in AArch64 Execution state.
- A 128-bit vector of byte, halfword, word, or doubleword elements. See Figure A1-1.
- A 64-bit vector of byte, halfword, word, or doubleword elements. See Figure A1-1.

For vectors, the element sizes are defined in Table A1-1, with the vector format described as:

- For a 128-bit vector: Vn.{2D, 4S, 8H, 16B}.
- For a 64-bit vector: Vn.{1D, 2S, 4H, 8B}.

Table A1-1 SIMD elements in AArch64 state

Mnemonic	Size
`B`	8 bits
`H`	16 bits
`S`	32 bits
`D`	64 bits

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure A1-1 SIMD vectors in AArch64 state

A1.4.1.2 Advanced SIMD vector formats in AArch32 state

Table A1-2 shows the available formats. Each instruction description specifies the data types that the instruction supports.

Table A1-2 Advanced SIMD data types in AArch32 state

Data type specifier	Meaning
`.<size>`	Any element of `<size>` bits
`.F<size>`	Floating-point number of `<size>` bits
`.I<size>`	Signed or unsigned integer of `<size>` bits
`.P<size>`	Polynomial over {0, 1} of degree less than `<size>`
`.S<size>`	Signed integer of `<size>` bits
`.U<size>`	Unsigned integer of `<size>` bits

Polynomial arithmetic over {0, 1} describes the polynomial data type.

The `.F16` data type is the half-precision data type selected by the FPSCR.AHP bit, see Half-precision floating-point formats.

The `.F32` data type is the Arm standard single-precision floating-point data type, see Single-precision floating-point format.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The instruction definitions use a data type specifier to define the data types appropriate to the operation. Figure A1-2 shows the hierarchy of the Advanced SIMD data types.

<table>
<tr><td rowspan="4">.8</td><td rowspan="2">.I8</td><td>.S8</td></tr>
<tr><td>.U8</td></tr>
<tr><td colspan="2">.P8</td></tr>
<tr><td colspan="2">-</td></tr>
<tr><td rowspan="5">.16</td><td rowspan="2">.I16</td><td>.S16</td></tr>
<tr><td>.U16</td></tr>
<tr><td colspan="2">.P16 †</td></tr>
<tr><td colspan="2">.F16</td></tr>
<tr><td colspan="2">.BF16</td></tr>
<tr><td rowspan="3">.32</td><td rowspan="2">.I32</td><td>.S32</td></tr>
<tr><td>.U32</td></tr>
<tr><td colspan="2">-</td></tr>
<tr><td rowspan="5">.64</td><td colspan="2">.F32</td></tr>
<tr><td rowspan="2">.I64</td><td>.S64</td></tr>
<tr><td>.U64</td></tr>
<tr><td colspan="2">.P64 ‡</td></tr>
<tr><td colspan="2">-</td></tr>
</table>

† Output format only. See `VMULL` instruction description.

‡ Available only if the Cyptographic Extension is implemented. See `VMULL` instruction description.

Figure A1-2 Advanced SIMD data type hierarchy in AArch32 state

For example, a multiply instruction must distinguish between integer and floating-point data types.

An integer multiply instruction that generates a double-width (long) result must specify the input data types as signed or unsigned. However, some integer multiply instructions use modulo arithmetic, and therefore do not have to distinguish between signed and unsigned inputs.

Figure A1-3 shows the Advanced SIMD vectors in AArch32 state.

Note

In AArch32 state, a pair of even and following odd numbered doubleword registers can be concatenated and treated as a single quadword register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure A1-3 Advanced SIMD vectors in AArch32 state

The AArch32 general-purpose registers support vectors formats for use by the SIMD instructions in the Base instruction set. Figure A1-4 shows these formats, that means that a general-purpose register can be treated as either 2 halfwords or 4 bytes.

Figure A1-4 Vector formatting in AArch32 state

A1.4.2 SVE vector formats

In an implementation that includes the AArch64 SVE instructions, an SVE register can hold one or more packed or unpacked elements, all of the same size and type. The combination of a register and an element size describes a vector of elements. The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the *Effective SVE vector length* of the register.

Vector element indexes are in the range 0 to (number of elements - 1). An index of 0 refers to bits 0 to (element size -1) of the vector.

For the definition of Effective SVE vector length, see Configurable SVE vector lengths.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.4.2.1 Scalable vector formats in AArch64 state

In AArch64 state, the SVE scalable vector registers can be referred to as Z*n*, where *n* is a value from 0 to 31.

The SVE scalable vector registers support the following data formats for loads, stores, and data-processing operations:

- Advanced SIMD data formats. Bits[127:0] of each Z*n* register hold the correspondingly numbered V*n* SIMD&FP register.
- A configurable-length vector of byte, halfword, word, doubleword, or quadword elements. See Figure A1-5. Also see Z0-Z31 in The AArch64 Application Level Programmers' Model.

The element sizes are defined in Table A1-3 with the vector format described as:

- Zn.{Q, D, S, H, B}.

Table A1-3 SVE elements in AArch64 state

Mnemonic	Element size
`B`	8 bits
`H`	16 bits
`S`	32 bits
`D`	64 bits
`Q`	128 bits

Figure A1-5 shows the vector formats of an SVE scalable vector register containing:

- A configurable-length vector, where the *Effective SVE vector length* is 256 bits.
- A fixed-length 128-bit Advanced SIMD vector.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure A1-5 SVE vectors, and Advanced SIMD vectors, in an SVE scalable vector register

A1.4.3 Half-precision floating-point formats

The Arm architecture supports two half-precision floating-point formats:

- IEEE half-precision, as described in the IEEE 754-2008 standard.
- Arm *alternative half-precision* format.

Note

BFloat16 is not a half-precision floating-point format, see BFloat16 floating-point format.

Both formats can be used for conversions to and from other floating-point formats. FPCR.AHP controls the format in AArch64 state and FPSCR.AHP controls the format in AArch32 state. FEAT_FP16 adds half-precision data-processing instructions, which always use the IEEE format. These instructions ignore the value of the relevant AHP field, and behave as if it has an Effective value of 0. All SVE half-precision data-processing instructions, including conversions, ignore the value of FPCR.AHP, and behave as if it has an Effective value of 0.

The description of IEEE half-precision includes Arm-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For more information, especially on the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For both half-precision floating-point formats, the layout of the 16-bit format is the same. The format is:

The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

0 < exponent < `0x1F`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-15)} \times (1.fraction)$

The minimum positive normalized number is 2^{-14}, or approximately 6.104×10^{-5}.

The maximum positive normalized number is $(2 - 2^{-10}) \times 2^{15}$, or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == `0x1F`.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-14} \times (0.fraction)$

The minimum positive denormalized number is 2^{-24}, or approximately 5.960×10^{-8}.

Half-precision denormalized numbers are not flushed to zero by default. When FEAT_FP16 is implemented, the FPCR.FZ16 bit controls whether flushing denormalized numbers to zero is enabled for half-precision data-processing instructions. For details, see Flushing denormalized numbers to zero.

exponent == `0x1F`

The value depends on which half-precision format is being used:

IEEE half-precision

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[9]:

bit[9] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[9] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

Alternative half-precision

The value is a normalized number and is equal to:

$-1^{S} \times 2^{16} \times (1.\text{fraction})$

The maximum positive normalized number is $(2-2^{-10}) \times 2^{16}$ or 131008.

A1.4.4 Single-precision floating-point format

The single-precision floating-point format is as defined by the IEEE 754 standard.

This description includes Arm-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word with the format:

31	30 … 23	22 … 0
S	exponent	fraction

The interpretation of the format depends on the value of the exponent field, bits[30:23]:

0 < exponent < `0xFF`

The value is a normalized number and is equal to:

$(-1)^{S} \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-23}) \times 2^{127}$, or approximately 3.403×10^{38}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 When S==0.

-0 When S==1.

These usually behave identically. In particular, the result is *equal* if +0 and -0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-149}, or approximately 1.401×10^{-45}.

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see Flushing denormalized numbers to zero.

exponent == `0xFF`

The value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[22]:

bit[22] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[22] == 1

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the *default NaN*, see Default NaN.

Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.5 Double-precision floating-point format

The double-precision floating-point format is as defined by the IEEE 754 standard. Double-precision floating-point is supported by both SIMD and floating-point instructions in AArch64 state, and only by floating-point instructions in AArch32 state.

This description includes IMPLEMENTATION SPECIFIC details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, nabs, and signed zeros, see the IEEE 754 standard.

A double-precision value is a 64-bit doubleword, with the format:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Double-precision values represent numbers, infinities, and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

0 < exponent < `0x7FF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-1023)} \times (1.fraction)$

The minimum positive normalized number is 2^{-1022}, or approximately 2.225×10^{-308}.

The maximum positive normalized number is $(2 - 2^{-52}) \times 2^{1023}$, or approximately 1.798×10^{308}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-1022} \times (0.fraction)$

The minimum positive denormalized number is 2^{-1074}, or approximately 4.941×10^{-324}.

Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details, see Flushing denormalized numbers to zero.

exponent == `0x7FF`

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

The value is an infinity. As for single-precision, there are two infinities:

+infinity When S==0.

-infinity When S==1.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

bit[51] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[51] == 1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the *default NaN*, see Default NaN.

Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.6 BFloat16 floating-point format

BFloat16 is a 16-bit floating-point storage format that inherits many of its properties and behaviors from the IEEE 754 single-precision format described in Single-precision floating-point format.

The BFloat16 format is:

0 < exponent < `0xFF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-127)} \times (1.fraction)$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-7}) \times 2^{127}$, or approximately 3.390×10^{38}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0.

-0 when S==1.

These usually behave identically. However, they yield different results in some circumstances. For example, the sign of the result produced as the result of multiplying by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer bitwise comparison of the two halfwords.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.fraction)$

The minimum positive denormalized number is 2^{-133}, or approximately 9.184×10^{-41}.

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see Flushing denormalized numbers to zero.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

exponent == `0xFF`

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[6]:

bit[6] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[6] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

BFloat16 values are 16-bit halfwords that software can convert to single-precision format, by appending 16 zero bits, so that single-precision arithmetic instructions can be used. A single-precision value can be converted to a BFloat16 value, either by:

- Truncating, by removing the least significant 16 bits.
- Using the BFloat16 conversion instructions, see Table C3-113.

See also:

- Summary of BFloat16 instruction behaviors.

A1.4.7 FP8 floating-point formats

The Arm architecture supports two 8-bit floating-point (FP8) formats:

- OFP8 E4M3 format.
- OFP8 E5M2 format.

These formats are used by FP8 instructions, which are instructions, where the input or output elements are in FP8 floating-point format. FPMR.F8S1 controls the format for inputs named *First FP8 input data stream* and FPMR.F8S2 controls the format for inputs named *Second FP8 input data stream*. The format of an FP8 output from an FP8 instruction is determined by FPMR.F8D field.

For more information, see *OCP 8-bit Floating Point Specification (OFP8)*.

A1.4.7.1 OFP8 E4M3 format

The OFP8 E4M3 format is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

(0 < exponent < `0xF`**) || (exponent ==** `0xF` **&& fraction <** `0x7`**)**

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-7)} \times (1.fraction)$

The minimum positive normalized number is 2^{-6}.

The maximum positive normalized number is $(2 - 2^{-2}) \times 2^8$.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-6} \times (0.fraction)$

The minimum positive denormalized number is 2^{-9}.

FP8 denormalized numbers are never flushed to zero.

exponent == `0xF` **&& fraction ==** `0x7`

The value is a *signaling NaN*.

An infinity is not supported.

A1.4.7.2 OFP8 E5M2 format

The OFP8 E5M2 format is:

0 < exponent < `0x1F`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-15)} \times (1.fraction)$

The minimum positive normalized number is 2^{-14}.

The maximum positive normalized number is $(2 - 2^{-2}) \times 2^{15}$.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-14} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-16}.

FP8 denormalized numbers are never flushed to zero.

exponent == `0x1F`

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[1]:

bit[1] == 0 The NaN is a signaling NaN. The sign bit can take any value, and bit[0] is 1.

bit[1] == 1 The NaN is a quiet NaN. The sign bit and bit[0] can take any value.

A1.4.8 Conversion between floating-point and fixed-point values

The Arm architecture supports the conversion of a scalar floating-point value to or from a signed or unsigned fixed-point value in a general-purpose register. Conversion instructions take an argument, #fbits, that specifies the number of fraction bits in the fixed-point number. That is, #fbits indicates that the general-purpose register holds a fixed-point number with fbits bits after the binary point, where fbits is in the range 1 to 64 for a 64-bit general-purpose register, or 1 to 32 for a 32-bit general-purpose register:

- For a 64-bit register X_d:
  - The integer part is X_d[63:#fbits].
  - The fractional part is X_d[(#fbits-1):0].
- For a 32-bit register W_d or R_d:
  - The integer part is W_d[31:#fbits] or R_d[31:#fbits].
  - The fractional part is W_d[(#fbits-1):0] or R_d[(#fbits-1):0].

These instructions can cause the following floating-point exceptions:

Invalid Operation When the floating-point input is NaN or Infinity or when a numerical value cannot be represented within the destination register.

Inexact When the numeric result differs from the input value.

Input Denormal When flushing denormalized numbers to zero is enabled and the denormal input is replaced by a zero, see Flushing denormalized numbers to zero and Input Denormal exceptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

An out of range fixed-point result is saturated to the destination size.

For more information, see Floating-point exceptions and exception traps.

A1.4.9 Polynomial arithmetic over {0, 1}

Some SIMD instructions that operate on SIMD&FP registers can operate on polynomials over {0, 1}, see Supported data types. The polynomial data type represents a polynomial in x of the form $b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ where b_k is bit[k] of the value.

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

- $0 + 0 = 1 + 1 = 0$.
- $0 + 1 = 1 + 0 = 1$.
- $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$.
- $1 \times 1 = 1$.

That is:

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

A64, A32, and T32 provide instructions for performing polynomial multiplication of 8-bit values.

- For AArch32, see VMUL (integer and polynomial) and VMULL (integer and polynomial).
- For AArch64, see PMUL and PMULL, PMULL2.

The Cryptographic Extension adds the ability to perform long polynomial multiplies of 64-bit values. See PMULL, PMULL2.

A1.4.9.1 Pseudocode description of polynomial multiplication

In pseudocode, polynomial addition is described by the EOR operation on bitstrings.

Polynomial multiplication is described by the `PolynomialMult()` function defined in A-profile Architecture Pseudocode.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5 Floating-point support

The architecture includes the following types of floating-point instructions:

- Scalar floating-point instructions that operate on the lowest numbered element of the SIMD&FP registers.
- Advanced SIMD floating-point instructions that operate on multiple elements of the SIMD&FP registers.
- If FEAT_SVE is implemented, AArch64 SVE instructions that operate on multiple elements of the SVE scalable vector registers, in which the SIMD&FP registers occupy the least significant 128 bits.
- If FEAT_SME is implemented, AArch64 SME instructions that operate on multiple elements of the SVE scalable vector registers and the ZA storage.

The architecture can provide the following levels of floating-point support:

- No floating-point support. This option is licensed only for implementations targeting specialized markets.
- Advanced SIMD and floating-point support.
- SVE, plus Advanced SIMD and floating-point support.
- SME and SVE, plus Advanced SIMD and floating-point support.

Note

All Armv8-A systems that support standard operating systems with rich application environments provide hardware support for Advanced SIMD and floating-point instructions. All Armv9-A systems that support standard operating systems with rich application environments also provide hardware support for SVE2 instructions. It is a requirement of the ARM Procedure Call Standard for AArch64, see *Procedure Call Standard for the Arm 64-bit Architecture*.

The architecture supports the following floating-point formats:

Table A1-4 Floating-point formats

	AArch32 state	AArch64 state
IEEE 754 half-precision	Conversion instructions to and from IEEE 754 half-precision.	
	Supported for data processing if FEAT_FP16 is implemented.	
Arm *alternative half-precision*	Conversion instructions to and from non-IEEE 754 Arm alternative half-precision.	
IEEE 754 single-precision	Supported for scalar data processing.	Supported for scalar and vector data processing.
	Supported for Advanced SIMD, with restrictions. See Arm standard floating-point input and output values.	
IEEE 754 double-precision	Supported for scalar data processing.	Supported for scalar and vector data processing.
	Not supported for Advanced SIMD.	
BFloat16	Supported if FEAT_AA32BF16 is implemented.	Supported if FEAT_BF16 is implemented.
OFP8 E4M3	Not supported.	Supported if FEAT_FP8 is implemented.
OFP8 E5M2	Not supported.	Supported if FEAT_FP8 is implemented.

For many floating-point instructions, there are configurable behaviors, such as:

- Configurable rounding modes. See Rounding.
- Configurable trapping of floating-point exceptions. See Floating-point exceptions and exception traps.
- Configurable non-IEEE 754 *Default NaN* behavior. See NaN handling and the Default NaN.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Configurable non-IEEE 754 flushing to zero of denormalized numbers. See Flushing denormalized numbers to zero.

Floating-point computation using AArch32 Advanced SIMD instructions remains unchanged from Armv7. T32 and A32 Advanced SIMD floating-point instructions always use Arm standard floating-point arithmetic and performs IEEE 754 floating-point arithmetic with the following restrictions:

- Denormalized numbers are flushed to zero, see Flushing denormalized numbers to zero.
- Only default NaNs are supported, see Default NaN.
- The Round to Nearest rounding mode is used.
- Untrapped floating-point exception handling is used for all floating-point exceptions.

If floating-point exception trapping is supported, then unless stated otherwise, when a floating-point exception is not trapped, that exception causes a cumulative status register bit to be set to 1 and the operation produces a default result. For more information, see Floating-point exceptions and exception traps.

In:

- AArch64 state, the Floating-point Control Register, FPCR, controls floating-point operation, and the Floating-point Status Register, FPSR, returns floating-point status information.
- AArch32 state, there is a single Floating-Point Status and Control Register, FPSCR, combining the FPCR and FPSR fields.

In AArch64 state, the PSTATE.{N,Z,C,V} condition flags are updated by floating-point comparison operations.

In AArch32 state, the FPSCR.{N,Z,C,V} condition flags are updated by floating-point comparison operations.

If FEAT_FlagM2 is implemented, the AArch64 instructions `AXFLAG` and `XAFLAG` convert between the Arm floating-point condition flags in PSTATE and an alternative format shown in Table C6-1.

Note

For additional details of AArch32 state floating-point support, see Advanced SIMD and floating-point support.

The remainder of this section contains:

- Instruction support.
- Floating-point standards, and terminology.
- Arm standard floating-point input and output values.
- Summary of BFloat16 instruction behaviors.
- Summary of FP8 instruction behaviors.
- Flushing denormalized numbers to zero.
- NaN handling and the Default NaN
- Rounding
- Floating-point exceptions and exception traps
- Alternate BFloat16 behaviors
- BFloat16 behaviors for instructions that compute sum-of-products

A1.5.1 Instruction support

The floating-point instructions support:

- Load and store for single elements and vectors of multiple elements.

  Note

  Single elements are also referred to as scalar elements.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Data processing on single and multiple elements.
- Complex number arithmetic.
- Floating-point conversion between different levels of precision.
- Conversion between floating-point and integer types.
- Floating-point rounding.
- Atomic floating-point in-memory operations.

For more information on floating-point instructions in AArch64 state, see A64 Instruction Set Overview.

For more information on floating-point instructions in AArch32 state, see The AArch32 Instruction Sets Overview.

A1.5.2 Floating-point standards, and terminology

The Arm architecture includes support for all the required features of ANSI/IEEE Std 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*, referred to as IEEE 754-2008. However, some terms in this Manual are based on the 1985 version of this standard, referred to as IEEE 754-1985:

- Arm floating-point terminology generally uses the IEEE 754-1985 terms. This section summarizes how IEEE 754-2008 changes these terms.
- References to IEEE 754 that do not include the issue year apply to either issue of the standard.

Table A1-5 shows how the terminology in this Manual differs from that used in IEEE 754-2008.

Table A1-5 Floating-point terminology

This manual	IEEE 754-2008
Normalized[a]	Normal
Denormal, or denormalized	Subnormal
Round towards Minus Infinity (RM)	roundTowardsNegative
Round towards Plus Infinity (RP)	roundTowardsPositive
Round towards Zero (RZ)	roundTowardZero
Round to Nearest (RN)	roundTiesToEven
Round to Nearest with Ties to Away	roundTiesToAway
Rounding mode	Rounding-direction attribute

a. *Normalized number* is used in preference to *normal number*, because of the other specific uses of *normal* in this Manual.

A1.5.3 Arm standard floating-point input and output values

The Arm architecture provides full IEEE 754 floating-point arithmetic support. In AArch32 state, floating-point operations performed using Advanced SIMD instructions are limited to *Arm standard floating-point operation*, regardless of the selected rounding mode in the FPSCR.

Arm standard floating-point arithmetic supports the following input formats defined by the IEEE 754 floating-point standard:

- Zeros.
- Normalized numbers.
- Denormalized numbers are flushed to 0 before floating-point operations, see Flushing denormalized numbers to zero.
- NaNs.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Infinities.

Arm standard floating-point arithmetic supports the Round to Nearest (roundTiesToEven) rounding mode defined by the IEEE 754 standard.

Arm standard floating-point arithmetic supports the following output result formats defined by the IEEE 754 standard:

- Zeros.
- Normalized numbers.
- Results with an absolute value less than the minimum normalized number are flushed to zero, see Flushing denormalized numbers to zero.
- NaNs produced in floating-point operations are always the default NaN, see Default NaN.
- Infinities.

Note

AArch64 Advanced SIMD floating-point arithmetic is performed using the rounding mode selected by the FPCR.

A1.5.4 Summary of BFloat16 instruction behaviors

BFloat16 instructions follow the floating-point behaviors shown in Table A1-6 and Table A1-7:

Table A1-6 Advanced SIMD and floating-point, and SVE, BFloat16 instruction behaviors

		Behaviors when the PE is in Streaming SVE mode	Behaviors when the PE is not in Streaming SVE mode
Advanced SIMD & FP	`BFCVT`	Behaves as: Floating-point behaviors in Streaming SVE mode.	Follows single-precision behaviors.
		This instruction also supports the Alternate BFloat16 behaviors.	
	`BFCVTN`, `BFCVTN2` `BFMLALB`, `BFMLALT`	If *legal* when executed in Streaming SVE mode[a], behave as Floating-point behaviors in Streaming SVE mode.	Follows single-precision behaviors.
		These instructions also support the Alternate BFloat16 behaviors.	
	`BFDOT`, `BFMMLA`	If legal when executed in Streaming SVE mode[a], behave as BFloat16 behaviors for instructions that compute sum-of-products.	Behave as BFloat16 behaviors for instructions that compute sum-of-products.

SVE	`BFADD`, `BFSUB`
	`BFMLA`, `BFMLS`,
	`BFMUL`
		Behave as Floating-point behaviors in Streaming SVE mode.	Follows single-precision behaviors.

	`BFCLAMP`
	`BFMAX`, `BFMAXNM`
	`BFMIN`, `BFMINNM`
		Behave as both:
		• Floating-point behaviors in Streaming SVE mode.
		• Additional behaviors described in the individual instruction descriptions.
			Follows both:
			• Single-precision behaviors.
			• Additional behaviors described in the individual instruction descriptions.

	`BFCVT`, `BFCVTNT`
	`BFMLALB`, `BFMLALT`
	`BFMLSLB`, `BFMLSLT`
		Behave as Floating-point behaviors in Streaming SVE mode.	Follows single-precision behaviors.

		These instructions also support the Alternate BFloat16 behaviors.	
	`BFDOT`	Behave as BFloat16 behaviors for instructions that compute sum-of-products.	
	`BFMMLA`	If legal when executed in Streaming SVE mode[a], behaves as BFloat16 behaviors for instructions that compute sum-of-products.	Behaves as BFloat16 behaviors for instructions that compute sum-of-products.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

a. These instructions generate an *SME illegal instruction exception* in *Streaming SVE mode* if FEAT_SME_FA64 is not implemented or not enabled at the current Exception level. See Streaming SVE mode and R_{ZFGJP}.

Table A1-7 SME BFloat16 instruction behaviors

SME	Behaviors in Streaming SVE mode		Behaviors when the PE is not in Streaming SVE mode
	ZA storage enabled	ZA storage disabled	
`BFADD`, `BFMLA`, `BFMLS`, `BFSUB` `BFMOPA` (non-widening) `BFMOPS` (non-widening)	Behave as Floating-point behaviors for instructions that target the SME ZA array.	Illegal[a]	Illegal[a]

`BFCLAMP`
`BFMAX`, `BFMAXNM`
`BFMIN`, `BFMINNM`
	Behave as both:
	• Floating-point behaviors in Streaming SVE mode.
	• Additional behaviors described in the instruction descriptions.
			Illegal[a]

`BFCVT` (single-precision to BFloat16), `BFCVTN`	Behave as Floating-point behaviors in Streaming SVE mode. These instructions also support the Alternate BFloat16 behaviors.		Illegal[a]
`BFDOT`, `BFVDOT` `BFMOPA` (widening) `BFMOPS` (widening)	Behave as BFloat16 behaviors for instructions that compute sum-of-products.	Illegal[a]	Illegal[a]
`BFMLAL` `BFMLSL`	Behave as Floating-point behaviors for instructions that target the SME ZA array.	Illegal[a]	Illegal[a]

a. The instructions generate an *SME illegal instruction exception*. See Streaming SVE mode and R_{ZFGJP}.

Unless otherwise specified, BFloat16 instructions follow the behaviors defined in all of:

- Flushing denormalized numbers to zero.
- Rounding.
- Floating-point exceptions and exception traps

See also:

- Streaming SVE mode.
- ZA storage.

A1.5.5 Summary of FP8 instruction behaviors

All FP8 instructions:

- Use the Round to Nearest (RN) mode, ignoring FPCR.RMode.
- Do not flush denormalized inputs and outputs to zero, as if FPCR.{FZ, FIZ, FZ16} are all 0.
- Generate Default NaN outputs, as if FPCR.DN is 1, unless otherwise specified.
- Honor FPCR.AH, if FEAT_AFP is implemented.
- Treat an unsupported FP8 input format, as defined in FPMR.F8S1 and FPMR.F8S2, as a CONSTRAINED UNPREDICTABLE choice of:
  - A signaling NaN.
  - Any of the supported FP8 formats.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Handle an unsupported FP8 output format, as defined in FPMR.F8D, by a CONSTRAINED UNPREDICTABLE choice of:
  - Setting the result to `0xff` and causing an Invalid Operation floating-point exception.
  - Generating the expected result of any of the supported FP8 formats.
- Do not cause an Input Denormal floating-point exception or update FPSR.IDC.

FP8 convert instructions:

- Produce the expected default result and update FPSR.{IOC, IXC, UFC} when they detect Invalid Operation, Inexact, or Underflow exceptional floating-point conditions.
- When the rounded result with unbounded exponent has an absolute value that is greater than the maximum normalized number for the output precision, or when an Infinity is converted from other floating-point formats to an FP8 format:
  - Generate one of the following outputs with the sign of the result:
    - When FPMR.OSC is 1, the maximum normalized number.
    - When FPMR.OSC is 0 and the output is in the OFP8 E4M3 format, a *signaling NaN*.
    - Otherwise, an Infinity.
  - Regardless of the value of FPMR.OSC, cause an Overflow and Inexact floating-point exception.
- When the PE is in Streaming SVE mode, disable trapped floating-point exceptions as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all zero.

FP8 multiplication instructions:

- Produce the expected default result and do not update FPSR.{IOC, IXC, UFC} when they detect Invalid Operation, Inexact, or Underflow exceptional floating-point conditions.
- When the rounded result with unbounded exponent has an absolute value that is greater than the maximum normalized number for the output precision:
  - Generate one of the following outputs with the sign of the result:
    - When FPMR.OSM is 1, the maximum normalized number.
    - When FPMR.OSM is 0, an Infinity.
  - Do not update FPSR.OFC.
- Disable trapped floating-point exceptions, as if FPCR.{IDE,IXE,UFE,OFE,DZE,IOE} are all zero.

A1.5.6 Flushing denormalized numbers to zero

For this section if FEAT_AFP is not implemented, the behavior is the same as if FPCR.AH == 0, FPCR.FIZ == 0, and FPCR.NEP == 0.

Calculations involving denormalized numbers and Underflow exceptions can reduce the performance of floating-point processing. For many algorithms, replacing the denormalized operands and intermediate results with zeros can recover this performance, without significantly affecting the accuracy of the final result. Arm floating-point implementations allow denormalized numbers to be flushed to zero to permit this optimization.

If a number `value` satisfies the condition `0 < Abs(value) < MinNorm`, it is treated as a denormalized number.

`MinNorm` is defined as follows:

- For OFP8 E4M3 numbers, `MinNorm` is 2^{-6}.
- For half-precision and OFP8 E5M2 numbers, `MinNorm` is 2^{-14}.
- For single-precision and BFloat16 numbers, `MinNorm` is 2^{-126}.
- For double-precision numbers, `MinNorm` is 2^{-1022}.

Flushing denormals to zero is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Enabling flushing of denormals to zero must be done with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application-dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not usually process denormalized numbers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On other algorithms, it can cause exceptions to occur and can seriously reduce the accuracy of the results of the algorithm.

For information about FP8 instruction behaviors, see Summary of FP8 instruction behaviors.

A1.5.6.1 Flushing denormalized inputs to zero

If flushing denormalized inputs to zero is enabled for an instruction and a data type, and an input to that instruction is a denormalized number of that data type, the input operand is flushed to zero, and its sign bit is not changed.

If a floating-point operation has a denormalized input that is flushed to zero, for all purposes within the instruction other than calculating Input Denormal floating-point exceptions, all inputs that are denormalized numbers are treated as though they were zero with the same sign as the input.

For floating-point instructions, if the instruction processes half-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If FPCR.FZ16 == 0, denormalized half-precision inputs are not flushed to zero.
- If FPCR.FZ16 == 1:
  - Denormalized half-precision inputs to data processing instructions are flushed to zero.
  - Denormalized half-precision inputs to convert instructions are not flushed to zero.

If FPCR.FIZ == 1, or both FPCR.AH == 0 and FPCR.FZ == 1, then for floating-point instructions other than `FABS` and `FNEG`, for all inputs other than half-precision, denormalized inputs are flushed to zero.

If FPCR.FZ == 0, for all inputs other than half-precision, FPCR.FZ does not cause denormalized inputs to be flushed to zero, although other factors might cause denormalized inputs to be flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.FIZ, all of the following instructions flush all input denormal numbers to zero:

- The BFloat16 instructions defined in Alternate BFloat16 behaviors.
- Single-precision and double-precision instructions: FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

A1.5.6.2 Flushing denormalized outputs to zero

If a denormalized output is flushed to zero, the output is returned as zero with the same sign bit as the denormalized output value.

If FPCR.AH == 0, the test for a denormalized number for the purpose of flushing the output to zero occurs before rounding.

If FPCR.AH == 1, the test for a denormalized number for the purpose of flushing the output to zero occurs after rounding using an unbounded exponent.

If FPCR.FZ == 0, then unless otherwise specified, for all outputs other than half-precision, FPCR.FZ does not cause denormalized outputs to be flushed to zero, although other factors might cause denormalized outputs to be flushed to zero.

If FPCR.FZ == 1, for all outputs other than half-precision, for floating-point instructions other than `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMIN`, and `BFMAX`, denormalized outputs are flushed to zero.

If FPCR.EBF == 0, `BFDOT`, `BFMMLA`, BFVDOT, BFMOPA (widening), and BFMOPS (widening) instructions unconditionally flush denormalized outputs to zero, as defined in BFloat16 behaviors for instructions that compute sum-of-products.

If FPCR.FZ16 == 0, denormalized half-precision outputs are not flushed to zero.

If FPCR.FZ16 == 1, for floating-point instructions other than `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMIN`, and `BFMAX`, denormalized half-precision outputs are flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.{FZ, FZ16}, denormalized outputs of `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMIN`, and `BFMAX`, are not flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.FZ, all of the following instructions flush all output denormal numbers to zero:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The BFloat16 instructions defined in Alternate BFloat16 behaviors.
- Single-precision and double-precision instructions: FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

If FPCR.FIZ == 1, for the atomic floating-point memory instructions defined in Atomic floating-point memory operations, denormalized BFloat16, single-precision, and double-precision outputs are flushed to zero.

A1.5.6.3 Flushing denormalized BFloat16 intermediate results to zero

BFloat16 arithmetic instructions `BFDOT`, `BFMMLA`, `BFVDOT`, `BFMOPA (widening)`, and `BFMOPS (widening)` in AArch64 state, and VDOT (by element), VDOT (vector), VMMLA in AArch32 state when working with BFloat16 inputs, convert BFloat16 input values to IEEE single-precision format, and calculate N-way dot-products, accumulating the products in single-precision accumulators. If one of these instructions processes an intermediate result that is a single-precision denormalized number:

- If the instruction is one of AArch64 `BFDOT`, `BFMMLA`, `BFVDOT`, `BFMOPA (widening)`, and `BFMOPS (widening)`, the intermediate result is unconditionally flushed to zero if FEAT_EBF16 is not implemented or when FPCR.EBF is 0. See BFloat16 behaviors for instructions that compute sum-of-products.
- If the instruction is one of AArch32 VDOT (by element), VDOT (vector), VMMLA, the intermediate result is unconditionally flushed to zero.

A1.5.7 NaN handling and the Default NaN

The IEEE 754 standard defines a NaN as a number with all exponent bits set to 1 and a nonzero number in the mantissa, and specifies that the sign bit of a NaN has no significance.

The Arm architecture additionally defines the following in Table A1-8:

- A *Default NaN*, compliant with the IEEE 754 standard.
- A BFloat16 NaN, that follows the encoding in BFloat16 floating-point format.
- OFP8 NaN, that follow the encodings in FP8 floating-point formats.

For a quiet NaN output derived from a signaling NaN operand, the most significant fraction bit is set to 1.

A PE is forbidden to generate a NaN whose value is strongly correlated to the values of non-NaN inputs as a speculative result of a floating-point calculation not involving NaN inputs.

A1.5.7.1 Default NaN

A Default NaN is encoded as described in Table A1-8.

Table A1-8 Default NaN encodings

	Half-precision, IEEE Format	Single-precision	Double-precision	BFloat16	OFP8 E4M3	OFP8 E5M2
Sign bit if FPCR.AH == 0	0	0	0	0	0	0
Sign bit if FPCR.AH == 1	1	1	1	1	1	1
Exponent	`0x1F`	`0xFF`	`0x7FF`	`0xFF`	`0xF`	`0x1F`
Fraction	Bit[9] == ‘1’, bits[8:0] == 0	Bit[22] == ‘1’, bits[21:0] == 0	Bit[51] == ‘1’, bits[50:0] == 0	Bit[6] == ‘1’, bits[5:0] == 0	Bits[2:0] == ‘111’	Bit[1] == ‘1’, bit[0] == ‘0’

When FPCR.DN == 1, if any input to a floating-point operation is a NaN, the output is a Default NaN, unless the instruction is one of the following:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMAX`, and `BFMIN`. For these, see the individual instruction descriptions for their Default NaN behavior.

The atomic floating-point memory instructions defined in Atomic floating-point memory operations generate only the Default NaN value, as if FPCR.DN == 1.

Floating-point instructions that detect an Invalid Operation condition for a reason other than one of its inputs being a NaN, produce a Default NaN as the result.

A1.5.7.2 NaN handling

The IEEE 754 standard does not specify which input NaN is used as the output NaN. Therefore, where the Arm architecture specifies which input NaN to use, this is an addition to the requirements in the IEEE 754 standard.

Depending on the operation, the exact value of a derived quiet NaN output might have both a different sign and different fraction bits from its source. See individual instruction descriptions for details.

A1.5.7.3 NaN propagation

For NaN propagation rules for:

- BFDOT and BFMMLA, see BFloat16 behaviors for instructions that compute sum-of-products.
- FP8 instructions, see Summary of FP8 instruction behaviors.

Otherwise, behavior is as described in this section.

If an output NaN is derived from one of the operands, how the input NaN propagates to the output depends on the instruction and the number of operands.

If an output NaN is derived from an input NaN and if the size of the output format is the same as the input format, then all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is derived from the input NaN.
- If the input NaN is a signaling NaN, the output NaN is derived as follows:
  - If the handling of a signaling NaN by the instruction detects an Invalid Operation condition, the output NaN is derived from a quietened version of the input NaN.
  - If the handling of a signaling NaN by the instruction does not detect an Invalid Operation condition, the output NaN is derived from the input NaN.

If an output NaN is derived from an input NaN and if the size of the output format is larger than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is zero-extended in the low-order bit to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN, except that the mantissa is zero-extended in the low-order bits and the exponent field is set to all ones.

If an output NaN is derived from an input NaN and if the size of the output format is smaller than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

For the following descriptions, the term “first operand” and “second operand” relate to the left-to-right ordering of the arguments of the pseudocode function that describes the operation.

If FPCR.DN == 0, for instructions that perform a floating-point operation, other than `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, NaN outputs that derive from NaN inputs are derived as follows:

- If all of the following apply, an instruction outputs a quiet NaN derived from the first signaling NaN operand:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - FPCR.AH == 0.
  - At least one operand is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, an instruction outputs a quiet NaN derived from the first NaN operand:
  - FPCR.AH == 0.
  - At least one operand is a NaN, but none of the operands is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, the output is a quiet NaN derived from the NaN operand:
  - FPCR.AH == 1.
  - The operation has two floating-point inputs.
  - The operation has only one NaN operand.
- If all of the following apply, the output is a NaN derived from the `<Zn>`, `<Vn>`, `<Hn>`, `<Sn>`, or `<Dn>` register, except for SVE FSUBR (vectors) and SVE FDIVR, for which the output is a NaN derived from the `<Zm>` register:
  - FPCR.AH == 1.
  - The operation has two floating-point inputs.
  - The operation has two NaN operands.
- If all of the following apply, the output is a NaN derived from the NaN held in the `<Zn>`, `<Vn>`, `<Hn>`, `<Sn>`, or `<Dn>` registers:
  - FPCR.AH == 1
  - The instruction is one of those shown in Table A1-9.
  - One of the following applies:
    - The operation has three NaN operands.
    - The operation has two NaN operands and the `<Zn>`, `<Vn>`, `<Hn>`, `<Sn>` or `<Dn>` register holds a NaN.
- If all of the following apply, the output is a NaN derived from the NaN held in the `<Zm>`, `<Vm>`, `<Hm>`, `<Sm>`, or `<Dm>` registers:
  - FPCR.AH == 1
  - The instruction is one of those shown in Table A1-9.
  - The operation has two NaN operands and the `<Zn>`, `<Vn>`, `<Hn>`, `<Sn>` or `<Dn>` register does not hold a NaN.

Table A1-9

Advanced SIMD&FP	`BFMLALB`, `BFMLALT`, `FCMLA`, `FMADD`, `FMLA`, `FMLAL`, `FMLAL2`, `FMLS`, `FMLSL`, `FMLSL2`, `FMSUB`, `FNMADD`, and `FNMSUB`.
SVE	`BFMLA`, `BFMLALB`, `BFMLALT`, `BFMLS`, `BFMLSLB`, `BFMLSLT`, `FCMLA`, `FMAD`, `FMLA`, `FMLALB`, `FMLALT`, `FMLS`, `FMLSLB`, `FMLSLT`, `FMSB`, `FNMAD`, `FNMLA`, `FNMLS`, `FNMSB`.

- When the operation has four or more floating-point inputs:
  - If at least one argument to `FPDot()` is a signaling NaN, a function outputs a quiet NaN derived from the first signaling NaN argument.
  - If at least one argument to `FPDot()` is a NaN, but none of the arguments is a signaling NaN, a function outputs a quiet NaN derived from the first NaN argument.

If FPCR.AH == 0, and an output NaN is derived from an input NaN, the pseudocode functions `FPAbs()`, `FPNeg()`, `BFNeg()`, `FPTrigMAdd()`, and `FPTrigSSel()` can change the sign of the NaN.

If FPCR.AH == 1, and an output NaN is derived from an input NaN, for all cases, the sign bit of the NaN is unchanged.

For `FMAX*`, `FMIN*`, `BFMAX*`, `BFMIN*`, `FCLAMP`, and `BFCLAMP`, the NaN handling is described in the instruction description.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.8 Rounding

The rounding mode specifies how the exact result of a floating-point operation is rounded to a value in the destination format.

The rounding mode is either determined by the rounding mode control field FPCR.RMode or by the instruction.

If FPCR.AH == 1, for any value of FPCR.RMode, the following instructions use Round to Nearest on outputs:

- The BFloat16 instructions defined in Alternate BFloat16 behaviors.
- Single-precision and double-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.
- Half-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

For information about FP8 instruction behaviors, see Summary of FP8 instruction behaviors.

FPCR.RMode can select one of:

- Round to Nearest (RN) mode.
- Round towards Plus Infinity (RP) mode.
- Round towards Minus Infinity (RM) mode.
- Round towards Zero (RZ) mode.

The following additional rounding behaviors are not selected by FPCR.RMode, but are used by some instructions:

- Round to Odd.
- Round to Nearest with Ties to Away.

A1.5.8.1 Round to Nearest (RN) mode

Round to Nearest rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity. The sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with an even least significant digit.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

A1.5.8.2 Round towards Plus Infinity (RP) mode

Round towards Plus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not less than the value before rounding. The result can be plus infinity.

A1.5.8.3 Round towards Minus Infinity (RM) mode

Round towards Minus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the number in the output format that is closest to and not greater than the value before rounding. The result can be minus infinity.

A1.5.8.4 Round towards Zero (RZ) mode

Round towards Zero rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not greater in absolute value than the value before rounding.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.8.5 Round to Nearest with Ties to Away

Round to Nearest with Ties to Away rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is calculated as follows:

- If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with the largest absolute value.
- If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Round to Nearest with Ties to Away rounding is used by FCVTAS (scalar), FCVTAS (vector), FCVTAU (scalar), FCVTAU (vector), FRINTA (scalar), and FRINTA (vector).

A1.5.8.6 Round to Odd

Round to Odd is not defined by IEEE 754.

Round to Odd rounds the exact result of a floating-point operation to a value that is representable in the destination format. If the result of the rounding is inexact, the least significant bit of the mantissa is forced to 1.

The following instructions use Round to Odd:

- BFloat16 instructions defined in BFloat16 behaviors for instructions that compute sum-of-products.
- FCVTXN, FCVTXN2, FCVTX, and FCVTXNT, for which Round to Odd rounding can avoid double rounding errors when a floating-point value is converted to a lower precision destination format through an intermediate precision format.

Example A1-1 Converting 64-bit floating-point format to 16-bit floating-point format

A 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value using the following steps:

1. Use an `FCVTXN` instruction to produce a 32-bit value.
2. Use another instruction with the required rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

A1.5.9 Floating-point exceptions and exception traps

Execution of a floating-point instruction, or execution of an Advanced SIMD, SVE, or SME instruction that performs floating-point operations, can generate an exceptional condition, called a *floating-point exception*.

Predicated SVE floating-point instructions only generate floating-point exceptions in response to floating-point operations performed on *Active elements*.

Note

In AArch64 state, an Advanced SIMD, SVE, or SME instruction that operates on floating-point values can perform multiple floating-point operations. Therefore, this section describes the handling of a floating-point exception on an *operation*, rather than on an *instruction*.

The architecture does not support asynchronous reporting of floating-point exceptions.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Overflow.
- Divide by Zero.
- Invalid Operation.

If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements about synchronous exception generation from that floating-point exception do not apply to the implementation.

Synchronous exception generation by floating-point exceptions is enabled using the FPCR as follows:

- For each floating-point exception that supports synchronous exception generation, the relevant control bit chosen from FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} is used to enable synchronous exception generation.
- For each floating-point exception that does not support synchronous exception generation, the relevant bit chosen from FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} is RAZ/WI.

For information about FP8 instruction behaviors, see Summary of FP8 instruction behaviors.

A1.5.9.1 Operations that do not generate floating-point exceptions

The following instructions do not generate any floating-point exceptions:

- The SVE exponential accelerator instruction, FEXPA.
- The SVE trigonometric select coefficient instruction, FTSSEL.
- The BFloat16 instructions defined in BFloat16 behaviors for instructions that compute sum-of-products.
- The SME instructions defined in Floating-point behaviors for instructions that target the SME ZA array.
- The instructions defined in Atomic floating-point memory operations.

If FPCR.AH is 1, all of the following instructions do not generate any floating-point exceptions regardless of their input values:

- The BFloat16 instructions defined in Alternate BFloat16 behaviors.
- Single-precision, double-precision and half-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

All the following apply to `FPAbs()`, `FPNeg()`, and `BFNeg`():

- They cannot generate floating-point exceptions.
- The floating-point behavior described in the Flushing denormalized numbers to zero does not apply to them.
- The floating-point behavior described in the section NaN handling and the Default NaN does not apply to them.
- When FPCR.AH is 1, the sign of a NaN input is not changed. When FPCR.AH is 0, the sign of a NaN input can be changed.

A1.5.9.2 Input Denormal exceptions

The cumulative floating-point exception bit FPSR.IDC, and the trap enable bit FPCR.IDE both relate to Input Denormal exceptions.

Input Denormal exceptions are not generated for denormalized half-precision inputs. For BFloat16 denormalized inputs and FP8 denormalized inputs, see Summary of BFloat16 instruction behaviors and Summary of FP8 instruction behaviors respectively. For single-precision and double-precision inputs:

- When FPCR.{AH, FZ, FIZ} is {0, 0, x}, Input Denormal exceptions are not generated.
- When FPCR.{AH, FZ, FIZ} is {0, 1, x}, an Input Denormal exception is generated when an input is flushed to zero.
- When FPCR.{AH, FZ, FIZ} is {1, x, 1}, Input Denormal exceptions are not generated.
- When FPCR.{AH, FZ, FIZ} is {1, x, 0}, an operation with a denormalized input generates an Input Denormal exception unless any of the following are true:
  - Another operand to the operation is a NaN.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The operation generates a Divide-by-Zero exception.
- The operation generates an Invalid Operation exception.
- The instruction that generated the operation was one of:
  - Advanced SIMD&FP convert from single-precision to BFloat16: BFCVT, BFCVTN, BFCVTN2.
  - Advanced SIMD&FP convert to integer: `FCVTAS`, `FCVTAU`, `FCVTMS`, `FCVTMU`, `FCVTNS`, `FCVTNU`, `FCVTPS`, `FCVTPU`, `FCVTZS`, `FCVTZU`.
  - Advanced SIMD&FP round to integral: `FRINT32X`, `FRINT32Z`, `FRINT64X`, `FRINT64Z`, `FRINTA`, `FRINTI`, `FRINTM`, `FRINTN`, `FRINTP`, `FRINTX`, `FRINTZ`.
  - Advanced SIMD&FP JavaScript convert to signed fixed-point: FJCVTZS.
  - SVE convert from single-precision to BFloat16: BFCVT, BFCVTNT.
  - SVE convert to integer: FCVTZS, FCVTZU.
  - SVE round to integral: FRINT<r>.
  - SME convert from single-precision to BFloat16: BFCVT, BFCVTN.
  - SME convert to integer: FCVTZS, FCVTZU.
  - SME round to integral: FRINTA, FRINTM, FRINTN, FRINTP.

When a denormalized input is flushed to zero, the occurrence of floating-point exceptions other than Input Denormal is determined by treating the input value that is flushed to zero as zero.

A1.5.9.3 Inexact exceptions

The cumulative floating-point exception bit FPSR.IXC and the trap enable bit FPCR.IXE both relate to Inexact exceptions.

If a denormalized output is flushed to zero, all of the following apply:

- If FPCR.AH is 1, an Inexact exception is generated.
- If FPCR.AH is 0, an Inexact exception is not generated.

If a result is not flushed to zero, and the result does not equal the result computed with unbounded exponent range and unbounded precision, then an Inexact exception is generated.

A1.5.9.4 Underflow exceptions

The cumulative floating-point exception bit FPSR.UFC, and the trap enable bit FPCR.UFE both relate to Underflow exceptions.

If FPCR.AH is 0, for the purpose of Underflow floating-point exception generation, a denormalized number is detected before rounding is applied.

If FPCR.AH is 1, for the purpose of Underflow floating-point exception generation, a denormalized number is detected after rounding with an unbounded exponent.

If the result of a floating-point operation is a denormalized number that is not flushed to zero, then:

- If the result is exact, an Underflow floating-point exception is generated only if FPCR.UFE is 1.
- If the result is inexact, an Underflow floating-point exception is always generated.

If the result of a floating-point operation is a denormalized number that is flushed to zero, then the Underflow floating-point exception is generated, however this floating-point exception is not trapped regardless of the value of FPCR.UFE.

A1.5.9.5 Overflow exceptions

The cumulative floating-point exception bit FPSR.OFC, and the trap enable bit FPCR.OFE both relate to Overflow exceptions.

If the output of an instruction rounded with an unbounded exponent is greater than the maximum normalized number for the output precision, an Overflow exception is generated.

If an untrapped Overflow exception is generated, the result is determined by the selected rounding behavior and the sign of the result before rounding, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Round to Nearest carries all overflows to infinity with the sign of the result before rounding, except for FP8 instructions, for which, see Summary of FP8 instruction behaviors.
- Round towards Plus Infinity carries negative overflows to the most negative finite number of the output precision, and carries positive overflows to plus infinity.
- Round towards Minus Infinity carries positive overflows to the largest finite number of the output precision, and carries negative overflows to minus infinity.
- Round towards Zero carries all overflows to the output precision's largest finite number with the sign of the result before rounding.
- Round to Nearest Ties to Away carries all overflows to infinity with the sign of the result before rounding.
- Round to Odd, when used by convert instructions, carries all overflows to the output precision's largest finite number with the sign of the result before rounding.
- Round to Odd, when used by some BFloat16 instructions, carries all overflows to infinity with the sign of the result before rounding.

A1.5.9.6 Divide by Zero exceptions

The cumulative floating-point exception bit FPSR.DZC, and the trap enable bit FPCR.DZE both relate to Divide by Zero exceptions.

If a floating-point operation divides a finite nonzero number by zero, a Divide by Zero exception is generated.

If a floating-point operation divides a finite nonzero number by zero, and the Divide by Zero exception is untrapped, the result is a correctly signed infinity.

A1.5.9.7 Invalid Operation exceptions

The cumulative floating-point exception bit FPSR.IOC, and the trap enable bit FPCR.IOE both relate to Invalid Operation exceptions.

For any floating-point instruction that performs a floating-point operation, if any of the following apply, the instruction generates an Invalid Operation exception:

- At least one operand is a signaling NaN.
- Subtracting two equal-magnitude infinities that are of the same sign.
- When FPCR.AH is 0, multiplying a zero by an infinity.
- Dividing a zero by a zero.
- Dividing an infinity by an infinity.
- Taking the square root of an operand that is less than zero.

When FPCR.AH is 1, a floating-point instruction multiplying a zero by an infinity:

- Generates an Invalid Operation exception if the instruction is one of the following fused multiply-add instructions and the addend is not a quiet NaN:
  - Advanced SIMD `FMLA`, `FMLAL`, `FMLAL2`, `FMLS`, `FMLSL`, `FMLSL2`.
  - SVE `FMLA`, `FMLALB`, `FMLALT`, `FMLS`, `FMLSLB`, `FMLSLT`, `BFMLA`, `BFMLS`.
- Generates an Invalid Operation exception otherwise.

If the input is one of: a quiet NaN, an infinity, or a number that overflows the values that can be represented in the output format, and if another exception is not generated to signal the condition, then a conversion from floating-point to either integer or fixed-point format, generates an Invalid Operation exception.

For the signaling compare instructions FCMPE and FCCMPE, if either of the source operands is any type of NaN, the instruction generates an Invalid Operation floating-point exception.

If FPCR.AH is 1, for FMAX (vector), FMAX (scalar), FMAXP (scalar), FMAXP (vector), FMAXV, FMIN (vector), FMIN (scalar), FMINP (scalar), FMINP (vector), FMINV, BFMAX, and BFMIN, if either input is any type of NaN, then an Invalid Operation floating-point exception is generated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.9.8 Handling floating-point exceptions

If an implementation supports synchronous exception generation for floating-point exceptions, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception and that is not at a lower Exception level than where the exception was generated.

If an implementation supports synchronous exception generation for floating-point exceptions in AArch64 state, all of the following apply:

- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception, except that an implementation is permitted to not restore the cumulative floating-point exception bits in the event of such an exception.
- When the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports one exception associated with one element that the instruction uses. The architecture does not specify which element is reported.

The `AArch64.FPTrappedException()` and `FPProcessException()` pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch64 state.

A1.5.9.9 Combinations of floating-point exceptions

More than one floating-point exception might occur on the same operation. The architecture permits only the following combinations of floating-point exceptions:

- Overflow with Inexact.
- Underflow with Inexact.
- If FPCR.AH is 0, Input Denormal with one or more of the other floating-point exception types.
- If FPCR.AH is 1, Input Denormal with Inexact, Underflow, or Overflow.

If two floating-point exceptions occur on the same operation, the Input Denormal exception is treated as highest priority and the Inexact exception is treated as lowest priority.

Some floating-point instructions specify more than one floating-point operation, this is indicated by the pseudocode descriptions of the instruction. In these cases, it is possible for one instruction to generate multiple exceptions. Multiple exceptions from one instruction are prioritized as follows:

- If an exception generating operation outputs a result that is used by a second exception generating operation, the exception generated by the operation that outputs the result is treated as higher priority than the exception generated by the second operation that uses the result.
- If exception generating operations do not use the outputs of other exception generating operations, it is CONSTRAINED UNPREDICTABLE which floating-point exception is treated as higher priority. The exception prioritized might differ between different instances of the same two floating-point exceptions being generated on the same operation during execution of the instruction.
- A trapped Underflow exception has priority over a trapped Inexact exception.

If none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative exception flag bit in the FPSR to be set to 1.

When FPCR.AH is 1, if the result of a floating-point operation is a denormalized number that is flushed to zero, both an untrapped Underflow floating-point exception and an Inexact floating-point exception are generated, which always sets the cumulative Underflow exception flag FPSR.UFC to 1 regardless of whether the Inexact floating-point exception is trapped. Otherwise, when a trapped floating-point exception is taken, it is IMPLEMENTATION DEFINED whether the FPSR is restored to the value it had immediately before the instruction that generated the trapped floating-point exception. If the IMPLEMENTATION DEFINED choice is that the FPSR is not restored, it is CONSTRAINED UNPREDICTABLE which trapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative exception flag bits being set to 1.

When a trapped floating-point exception is taken, then in the ESR_ELx, all of the following apply:

- The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
- If any other untrapped floating-point exceptions are generated by the same operation, each untrapped exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is CONSTRAINED UNPREDICTABLE whether the floating-point exception trapped bit is set to 1.

The architectural requirements for floating-point exception prioritization apply only to multiple floating-point exceptions generated on the same element of an Advanced SIMD, SVE or SME operation. For trapped floating-point exceptions from Advanced SIMD, SVE or SME instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction.

A1.5.10 Alternate BFloat16 behaviors

The behaviors in this section apply to the following instructions:

- Advanced SIMD&FP: `BFCVT`, `BFCVTN`, `BFCVTN2`, `BFMLALB`, `BFMLALT`.
- SVE: `BFCVT`, `BFCVTNT`, `BFMLALB`, `BFMLALT`, `BFMLSLB`, `BFMLSLT`.
- SME: `BFCVT` (single-precision to BFloat16), `BFCVTN`.

Table A1-6 and Table A1-7 describe when these instructions are legal.

When FPCR.AH is 1, the instructions:

- Produce the expected IEEE 754 default result but do not update the FPSR cumulative exception flag bits.
- Disable trapped floating-point exceptions, as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all 0.
- Use Round to Nearest Even, ignoring FPCR.RMode.
- Flush denormalized inputs and outputs to zero, as if FPCR.{FZ, FIZ} is {1, 1}.

A1.5.11 BFloat16 behaviors for instructions that compute sum-of-products

The behaviors in this section apply to the following instructions:

- Advanced SIMD: `BFDOT` and `BFMMLA`.
- SVE: `BFDOT` and `BFMMLA`.
- SME: `BFMOPA` (widening) and `BFMOPS` (widening), and SME2 `BFDOT` and `BFVDOT`.

Table A1-6 and Table A1-7 describe when these instructions are legal.

When FPCR.EBF is 0, the instructions:

- Perform unfused two-way sum-of-products for each pair of adjacent BFloat16 elements in the source vectors, with rounding of all intermediate products and sums.
- Ignore FPCR.RMode and use non-IEEE 754 Round to Odd behavior. See Round to Odd.
- Flush denormalized inputs and outputs to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Disable alternative floating-point behaviors, as if FPCR.AH is 0.

When FPCR.EBF is 1, the instructions:

- Perform a fused two-way sum-of-products for each pair of adjacent BFloat16 elements in the source vectors, without rounding of the intermediate products, but rounding the single-precision sum before addition to the single-precision accumulator element.
- Honor FPCR.RMode, supporting all four IEEE 754 rounding modes.
- Honor FPCR.{FZ, FIZ}.
- Honor FPCR.AH.

Regardless of the value of FPCR.EBF, the instructions:

- Produce the expected IEEE 754 single-precision default result but do not update the FPSR cumulative exception flag bits.
- Disable trapped floating-point exceptions, as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Generate an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.
- Generate an intermediate sum-of-products that is a Default NaN when any of the following are true:
  - Any multiplier input is a NaN.
  - Any product is infinity x 0.0.
  - Both products are infinity of differing signs.
- Generate a Default NaN, as if FPCR.DN is 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.6 The Arm memory model

The Arm memory model supports:

- Generating an exception on an unaligned memory access.
- Restricting access by applications to specified areas of memory.
- Translating *virtual addresses* (VAs) provided by executing instructions to *physical addresses* (PAs).
- Altering the interpretation of multi-byte data between big-endian and little-endian.
- Controlling the order of accesses to memory.
- Controlling caches and address translation structures.
- Synchronizing access to shared memory by multiple PEs.
- Barriers that control and prevent speculative access to memory.

VA support depends on the Execution state, as follows:

AArch64 state

Supports 64-bit virtual addressing, with the Translation Control Register determining the supported VA range. Execution at EL1 and EL0 supports two independent VA ranges, each with its own translation controls.

AArch32 state

Supports 32-bit virtual addressing, with the Translation Control Register determining the supported VA range. For execution at EL1 and EL0, system software can split the VA range into two subranges, each with its own translation controls.

The supported PA space is IMPLEMENTATION DEFINED, and can be discovered by system software.

Regardless of the Execution state, the *Virtual Memory System Architecture* (VMSA) can translate VAs to blocks or pages of memory anywhere within the supported PA space.

For more information, see:

For execution in AArch64 state

- The AArch64 Application Level Memory Model.
- The AArch64 System Level Memory Model.
- The AArch64 Virtual Memory System Architecture.

For execution in AArch32 state

- The AArch32 Application Level Memory Model.
- The AArch32 System Level Memory Model.
- The AArch32 Virtual Memory System Architecture.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.7 Reliability, Availability, and Serviceability

A1.7.1 Introduction

I_{LMHPD} RAS are three aspects of the *dependability* of a system:

- Reliability, that is, the continuity of correct service.
- Availability, that is, the readiness for correct service.
- Serviceability, that is, the ability to undergo modifications and repairs.

I_{HWHJM} RAS techniques reduce unplanned outages because:

- Transient errors can be detected and corrected before they cause application or system failure.
- Failing components can be identified and replaced.
- Failure can be predicted ahead-of-time to allow replacement during planned maintenance.

A1.7.2 Faults, errors, and failures

R_{NVNNC} Correct service is delivered when the service implements the system function.

I_{QWSVK} Correct service might include:

- Producing correct results.
- Producing results within the time allotted to the task.
- Not divulging secret or secure information.

R_{KRTSC} For the purpose of describing the RAS Extension and RAS System Architecture, deviation from correct service is defined using the following terms:

- A failure is the event of deviation from correct service. This includes data corruption, data loss, and service loss.
- An error is the deviation from correct service. An incorrect value that has an error is corrupt.
- A fault is the cause of the error.

R_{JNBDX} Errors that are present but not detected are latent errors or undetected errors.

I_{TNQPK} In a system with no error detection, all errors are latent errors and are silently propagated by components until they are either masked or cause failure.

I_{GRYKV} The severity of a failure can range from minor to catastrophic:

- The harmful consequences of a minor failure are of a similar cost to the benefits provided by correct service delivery.
- The harmful consequences of a catastrophic failure are orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

I_{NMGPQ} There are many sources of faults in a system, including both software and hardware faults:

- *Hardware faults* originate in, or affect, hardware.
- *Software faults* affect software, that is programs or data.

The RAS Extension and RAS System Architecture primarily address errors produced from hardware faults. These fall into two main areas:

- Transient faults.
- Non-transient or *persistent faults*.

A1.7.3 General taxonomy of errors

A1.7.3.1 Error detection

R_{FHXWP} When a component accesses memory or other state, an error might be detected in that memory or state.

I_{WKPVR} The error might be corrected or deferred by the component, or signaled to another component as either a Deferred error or a Detected error.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.7.3.2 Error propagation

R_{LRZDN} A transaction occurs when a producer of the transaction passes a value or other signal to a consumer of the transaction.

I_{VYCCX} Transactions are part of the service provided by the producer for the consumer.

I_{RHGDL} In many protocols and service interface definitions, a high-level transaction consists of a sequence of operations, for instance between a *Requester* and a *Completer*.

For the purposes of this manual, the most basic form of a unidirectional transfer between a *producer* and *consumer* is considered as a transaction.

That is, each one of the sequence of operations is considered a separate transaction. For some operations, such as a request, the Requester is producer and the Completer is the consumer. For other operations, such as a response, the Completer is producer and the Requester is the consumer.

R_{SKZZG} An error is propagated by the producer of a transaction when the service interface is incorrect because of the error. The error is propagated to the consumer.

R_{CHCCV} An error is propagated by deviations from correct service, including when any of the following occurs that would not have been permitted to occur had the fault not been activated:

- A corrupt value is passed from producer to consumer.
- A transaction or other operation occurs that should not have occurred.
- A transaction or other operation that should have occurred does not occur.
- A loss of uniprocessor semantics or any other loss of coherency in a multiprocessor coherent system is observed.
- Changing the timing and/or order of transactions or other operations such that the timing and/or order of those transactions or operations is incorrect. In this case, the service interface defines acceptable timings and/or orders for transactions and other operations.

R_{YFBRV} The service interface for a transaction might include means to signal that the transaction is propagating either of the following:

- A Detected error.
- A Deferred error.

R_{BHWVX} An error is silently propagated by the producer of a transaction if the consumer of the transaction cannot detect the error and consumes an undetected error because of the transaction. This might be because of one of the following:

- The error is present on the transaction, but was not detected by the producer. The error is silently propagated by the producer.
- The error is present on the transaction, but was not signaled to the consumer as an error. For example, a corrupt value was passed in the transaction with no indication that it was corrupt. The error is silently propagated by the producer.

R_{FPBYS} A latent, possibly detectable, error is silently propagated by the consumer of an otherwise correct transaction if the transaction causes the error to become undetectable.

Example A1-2

A partial write to a protection granule removes poison, leaving the unchanged portion of the location corrupt. To implement a partial write, the consumer logically reads the current value of the location, modifies the value, and then writes the modified value back. These are internal transactions in the consumer that silently propagate the error. In this example there was no error at the producer nor on the transaction.

I_{QXRLB} Errors might be propagated by components in a system until one of the following occurs:

- They are masked and do not affect the outcome of the system.
  The error might be masked because a corrupt value is discarded or overwritten, or the error is detected and removed.
- They affect the service interface of the system and possibly cause failure. If the error has been silently propagated to the service interface, then:
  — This is a Silent Data Corruption, SDC.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The rate of such failures, measured as the number of failures per billion device-hours of operation, is called the Silent Data Corruption Failure-in-Time rate, SDC FIT rate.

Alternatively, the error might have been detected, causing the system to invoke error handling and recovery. For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

A1.7.3.3 Infected and poisoned

R_{KNHWB} The state of a component becomes infected when the component consumes an Uncorrected error that updates the state.

R_{TZBSW} A value is poisoned in the state of a component if it is marked as being in error, such that a subsequent access of the state will detect the value is so marked and is treated as a Detected error.

I_{YBMFK} Poison is used to defer an error.

A1.7.3.4 Containable and uncontainable

R_{DXQRD} An undetected error is uncontained at the component that failed to detect it.

R_{RJYRQ} A silently propagated error is uncontained at the component that silently propagated it.

R_{GJQNR} A Detected Uncorrected error is uncontainable at the component if it might be uncontained at the component. A Detected Uncorrected error is containable at the component if it is not uncontainable at the component. If the component cannot determine whether a Detected Uncorrected error is uncontainable at the component or containable at the component, then the component treats the Detected Uncorrected error as uncontainable at the component.

I_{MRDMR} An error that is uncontainable at the component might be containable at the system level.

I_{NWZGB} Reporting an error as containable allows software to contain the error. This does not mean that hardware has contained the error.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter A2
A-profile Architecture Extensions

This chapter introduces the Arm A-profile architecture extensions. It contains the following sections:

- About the A-profile architecture extensions.
- Armv8-A architecture extensions.
- Armv9-A architecture extensions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.1 About the A-profile architecture extensions

An architecture extension defines a set of features that, when implemented, may make an implementation compliant with the extension.

Each feature description includes:

- A feature name.
- A brief description of the feature.
- The Execution state the feature is supported in.
- Whether the implementation of the feature is mandatory or OPTIONAL.
- Dependencies on the implementation of other features if any.
- The register field that identifies the presence of the feature.

A2.1.1 Permitted implementation of subsets of the A-profile architectural features

Arm regularly introduces new features to the architecture. When a feature is introduced, the architecture specifies when the feature can be implemented. Typically, a feature introduced as part of Armv8.*x* or Armv9.*x* is permitted to be built in Armv8.(*x*-1) or Armv9.(*x*-1), subject only to those constraints that require that certain features be implemented together.

See the individual feature descriptions for details of when each feature is permitted or required to be built.

A2.1.2 Withdrawn architecture features

This section lists features previously described in the architecture but subsequently withdrawn. Features listed in this section are not permitted to be included in any version of the architecture:

FEAT_TME, Transactional Memory Extension

FEAT_TME introduced a set of instructions to support hardware transaction memory, which means a group of instructions can appear to be collectively executed as a single atomic operation.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.2 Armv8-A architecture extensions

A2.2.1 The Armv8.0 architecture extension

The original Armv8-A architecture is called Armv8.0. It contains mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.0 compliant if it includes all of the Armv8.0 architectural features that are mandatory.

An Armv8.0 compliant implementation can additionally include:

- Armv8.0 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.1, subject only to those constraints that require that certain features be implemented together.

FEAT_AA32, PE supports AArch32.

FEAT_AA32 is OPTIONAL.

If FEAT_AA32 is implemented, then FEAT_AA32EL0 is implemented.

FEAT_AA32EL0, Support for AArch32 at EL0

FEAT_AA32EL0 is OPTIONAL from Armv8.0.

If FEAT_AA32EL0 is implemented, then FEAT_EL0 is implemented.

If FEAT_AA32EL0 is implemented, then FEAT_AA32 is implemented.

FEAT_AA32EL1, Support for AArch32 at EL1

FEAT_AA32EL1 is OPTIONAL from Armv8.0.

If Armv9.0 is implemented, then FEAT_AA32EL1 is not implemented.

If FEAT_AA32EL1 is implemented, then FEAT_AA32EL0 is implemented.

If FEAT_AA32EL1 is implemented, then FEAT_EL1 is implemented.

FEAT_AA32EL2, Support for AArch32 at EL2

FEAT_AA32EL2 is OPTIONAL from Armv8.0.

If FEAT_AA32EL2 is implemented, then FEAT_AA32EL1 is implemented.

If FEAT_AA32EL2 is implemented, then FEAT_EL2 is implemented.

FEAT_AA32EL3, Support for AArch32 at EL3

FEAT_AA32EL3 is OPTIONAL from Armv8.0.

If FEAT_AA32EL3 is implemented, then FEAT_AA32EL1 is implemented.

When FEAT_AA32EL3 and FEAT_EL2 are implemented, FEAT_AA32EL2 is implemented.

If FEAT_AA32EL3 is implemented, then FEAT_EL3 is implemented.

FEAT_AA64, PE uses AArch64 after last reboot

FEAT_AA64 is OPTIONAL from Armv8.0.

If FEAT_AA64 is implemented, then FEAT_AA64EL0, FEAT_AA64EL1, FEAT_AA64EL2, or FEAT_AA64EL3 is implemented.

When FEAT_AA64EL0, FEAT_AA64EL1, FEAT_AA64EL2, or FEAT_AA64EL3 is implemented, FEAT_AA64 is implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AA64EL0, Support for AArch64 at EL0

FEAT_AA64EL0 is OPTIONAL from Armv8.0.

FEAT_AA64EL0 is mandatory from Armv9.0.

If FEAT_AA64EL0 is implemented, then FEAT_AA64EL1 is implemented.

If FEAT_AA64EL0 is implemented, then FEAT_EL0 is implemented.

FEAT_AA64EL1, Support for AArch64 at EL1

FEAT_AA64EL1 is OPTIONAL from Armv8.0.

FEAT_AA64EL1 is mandatory from Armv9.0.

If FEAT_AA64EL1 is implemented, then FEAT_AA64EL0 is implemented.

When FEAT_AA64EL1 and FEAT_EL2 are implemented, FEAT_AA64EL2 is implemented.

When FEAT_AA64EL1 and FEAT_EL3 are implemented, FEAT_AA64EL3 is implemented.

If FEAT_AA64EL1 is implemented, then FEAT_EL1 is implemented.

FEAT_AA64EL2, Support for AArch64 at EL2

FEAT_AA64EL2 is OPTIONAL from Armv8.0.

If FEAT_AA64EL2 is implemented, then FEAT_AA64EL1 is implemented.

When FEAT_AA64EL2 and FEAT_EL3 are implemented, FEAT_AA64EL3 is implemented.

If FEAT_AA64EL2 is implemented, then FEAT_EL2 is implemented.

FEAT_AA64EL3, Support for AArch64 at EL3

FEAT_AA64EL3 is OPTIONAL from Armv8.0.

If FEAT_AA64EL3 is implemented, then FEAT_AA64EL1 is implemented.

When FEAT_AA64EL3 and FEAT_EL2 are implemented, FEAT_AA64EL2 is implemented.

If FEAT_AA64EL3 is implemented, then FEAT_EL3 is implemented.

FEAT_AES, Advanced SIMD AES instructions

FEAT_AES provides the AES* instructions to support AES encryption and decryption, AESD, AESE, AESMC, and AESIMC.

This feature is supported in both AArch64 and AArch32 states.

FEAT_AES is OPTIONAL from Armv8.0.

If FEAT_AES is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_AES:

- ID_AA64ISAR0_EL1.AES.

FEAT_ASID16, 16 bit ASID

FEAT_ASID16 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_ASID16:

- ID_AA64MMFR0_EL1.ASIDBits.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AdvSIMD, Advanced SIMD Extension

FEAT_AdvSIMD includes support for the SISD and SIMD operations. All Armv8-A systems that support standard operating systems with rich application environments also provide hardware support for Advanced SIMD instructions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_AdvSIMD is OPTIONAL from Armv8.0.

If FEAT_AdvSIMD is implemented, then FEAT_FP is implemented.

The following fields identify the presence of FEAT_AdvSIMD:

- ID_AA64PFR0_EL1.AdvSIMD.
- EDPFR.AdvSIMD.

For more information, see:

- Supported data types.
- A64 Advanced SIMD and Floating-point Instruction Descriptions.
- Advanced SIMD and floating-point instructions.
- T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions.

FEAT_BigEnd, Support for big-endian at EL1 and above

FEAT_BigEnd is OPTIONAL from Armv8.0.

If FEAT_BigEnd is implemented, then FEAT_BigEndEL0 is implemented.

If FEAT_MixedEnd is implemented, then FEAT_BigEnd is implemented.

FEAT_BigEndEL0, Support for big-endian at EL0

FEAT_BigEndEL0 is OPTIONAL from Armv8.0.

If FEAT_MixedEndEL0 is implemented, then FEAT_BigEndEL0 is implemented.

FEAT_CRC32, CRC32 instructions

FEAT_CRC32 introduces the support for the CRC32* instructions that perform cyclic redundancy check calculations.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CRC32 is OPTIONAL from Armv8.0.

FEAT_CRC32 is mandatory from Armv8.1.

The following fields identify the presence of FEAT_CRC32:

- ID_AA64ISAR0_EL1.CRC32.
- ID_ISAR5_EL1.CRC32.
- ID_ISAR5.CRC32.

FEAT_CSV2, Cache Speculation Variant 2

FEAT_CSV2 introduces a mechanism to identify if hardware cannot disclose information about whether any prediction resource, such as branch target, data-value, or cache prefetch, trained in one hardware described context can control speculative execution in a different hardware described context.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CSV2 is OPTIONAL from Armv8.0.

FEAT_CSV2 is mandatory from Armv8.5.

The following fields identify the presence of FEAT_CSV2:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64PFR0_EL1.CSV2.
- ID_PFR0_EL1.CSV2.
- ID_PFR0.CSV2.

For more information, see:

- For AArch64, 'Restrictions on the effects of speculation'.
- 'AArch32 restrictions on the effects of speculation'.

FEAT_CSV2_1p1, Cache Speculation Variant 2

For FEAT_CSV2_1p1, within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

FEAT_CSV2_1p1 does not support the SCXTNUM_ELx registers.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CSV2_1p1 is OPTIONAL from Armv8.0.

If FEAT_CSV2_1p1 is implemented, then FEAT_CSV2 is implemented.

The following fields identify the presence of FEAT_CSV2_1p1:

- ID_AA64PFR1_EL1.CSV2_frac.
- ID_AA64PFR0_EL1.CSV2.
- ID_PFR0_EL1.CSV2.
- ID_PFR0.CSV2.

For more information, see:

- For AArch64, 'Restrictions on the effects of speculation'.
- 'AArch32 restrictions on the effects of speculation'.

FEAT_CSV2_1p2, Cache Speculation Variant 2 version 1.2

For FEAT_CSV2_1p2, within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

FEAT_CSV2_1p2 introduces the SCXTNUM_ELx registers, but the contexts do not include the SCXTNUM_ELx register contexts.

This feature is supported in AArch64 state only.

FEAT_CSV2_1p2 is OPTIONAL from Armv8.0.

If FEAT_CSV2_1p2 is implemented, then FEAT_CSV2_1p1 is implemented.

The following fields identify the presence of FEAT_CSV2_1p2:

- ID_AA64PFR1_EL1.CSV2_frac.
- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, 'Restrictions on the effects of speculation'.
- 'AArch32 restrictions on the effects of speculation'.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_CSV2_2, Cache Speculation Variant 2 version 2

FEAT_CSV2_2 introduces the SCXTNUM_ELx registers, which provide a number that can be used to separate out different context numbers within their respective Exception levels for the purpose of protecting against speculative attacks through prediction resources.

This feature is supported in AArch64 state only.

FEAT_CSV2_2 is OPTIONAL from Armv8.0.

If FEAT_CSV2_2 is implemented, then FEAT_CSV2 is implemented.

If FEAT_CSV2_2 is implemented, then FEAT_CSV2_1p1 is not implemented.

The following field identifies the presence of FEAT_CSV2_2:

- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, 'Restrictions on the effects of speculation'.
- 'AArch32 restrictions on the effects of speculation'.

FEAT_CSV2_3, Cache Speculation Variant 2 version 3

FEAT_CSV2_3 introduces a mechanism to identify if hardware cannot disclose information about whether prediction resources based on branch history trained in one hardware described context can control speculative execution in a different hardware described context.

This feature is supported in AArch64 state only.

FEAT_CSV2_3 is OPTIONAL from Armv8.0.

If FEAT_CSV2_3 is implemented, then FEAT_CSV2_2 is implemented.

The following field identifies the presence of FEAT_CSV2_3:

- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, 'Restrictions on the effects of speculation'.
- 'AArch32 restrictions on the effects of speculation'.

FEAT_Crypto, Cryptographic Extension

The Arm Cryptographic Extension provides instructions for the acceleration of encryption and decryption. The presence of the Cryptographic Extension in an implementation is subject to export license controls.

The Cryptographic Extension is an extension of the SIMD support and operates on the vector register file. It also provides multiply instructions that operate on long polynomials.

In an implementation that supports both AArch64 state and AArch32 state, FEAT_AES, FEAT_PMULL, FEAT_SHA1 and FEAT_SHA256 provide the same functionality in both states.

This feature is supported in both AArch64 and AArch32 states.

FEAT_Crypto is OPTIONAL.

If FEAT_Crypto is implemented, then FEAT_AES or FEAT_SHA1 is implemented.

In an Armv8.2 implementation, if FEAT_Crypto is implemented, FEAT_PMULL, FEAT_SHA256, FEAT_SM3, or FEAT_SM4 is implemented.

For more information, see:

- The Cryptographic Extension in AArch64 state.
- The Cryptographic Extension in AArch32 state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_DoubleLock, Double Lock

FEAT_DoubleLock is the mnemonic used for the OS Double Lock.

FEAT_DoubleLock is OPTIONAL from Armv8.0.

If Armv9.0 is implemented, then FEAT_DoubleLock is not implemented.

The following fields identify the presence of FEAT_DoubleLock:

- ID_AA64DFR0_EL1.DoubleLock.
- DBGDEVID.DoubleLock.

FEAT_EL0, Support for execution at EL0

FEAT_EL0 is mandatory from Armv8.0.

If FEAT_EL0 is implemented, then FEAT_AA32EL0 or FEAT_AA64EL0 is implemented.

FEAT_EL1, Support for execution at EL1

FEAT_EL1 is mandatory from Armv8.0.

If FEAT_EL1 is implemented, then FEAT_AA32EL1 or FEAT_AA64EL1 is implemented.

FEAT_EL2, Support for execution at EL2

FEAT_EL2 is OPTIONAL from Armv8.0.

If FEAT_EL2 is implemented, then FEAT_AA32EL2 or FEAT_AA64EL2 is implemented.

FEAT_EL3, Support for EL3

FEAT_EL3 is OPTIONAL from Armv8.0.

If FEAT_EL3 is implemented, then FEAT_AA32EL3 or FEAT_AA64EL3 is implemented.

FEAT_ETMv4, Embedded Trace Macrocell version 4

FEAT_ETMv4 indicates support for the Embedded Trace Macrocell architecture ETMv4.

FEAT_ETMv4 is OPTIONAL from Armv8.0.

If Armv9.0 is implemented, then FEAT_ETMv4 is not implemented.

If FEAT_ETMv4 is implemented, then FEAT_TRC_SR or FEAT_TRC_EXT is implemented.

For more information, see the Arm® Embedded Trace Macrocell Architecture Specification, ETMv4(ARM IHI 0064).

FEAT_ETS2, Enhanced Translation Synchronization

FEAT_ETS2 introduces support for enhanced memory access ordering requirements for translation table walks.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ETS2 is OPTIONAL from Armv8.0.

FEAT_ETS2 is mandatory from Armv8.8.

The following fields identify the presence of FEAT_ETS2:

- ID_AA64MMFR1_EL1.ETS.
- ID_MMFR5_EL1.ETS.
- ID_MMFR5.ETS.

For more information, see:

- Ordering requirements defined by the formal concurrency model.
- Ordering of memory accesses from translation table walks.
- Ordering of translation table walks.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FP, Floating Point extensions

FEAT_FP includes support for single-precision and double-precision floating-point types. All Armv8-A systems that support standard operating systems with rich application environments also provide hardware support for floating-point instructions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_FP is OPTIONAL from Armv8.0.

If FEAT_FP is implemented, then FEAT_AdvSIMD is implemented.

The following fields identify the presence of FEAT_FP:

- ID_AA64PFR0_EL1.FP.
- EDPFR.FP.

For more information, see:

- Supported data types.
- Floating-point support.
- A64 Advanced SIMD and Floating-point Instruction Descriptions.
- Advanced SIMD and floating-point instructions.
- T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions.

FEAT_IVIPT, The IVIPT Extension

FEAT_IVIPT describes any permitted instruction cache implementation. This includes Virtual Index, Physical tag (VIPT) cache policy and Physical Index, Physical Tag (PIPT) cache policy.

FEAT_IVIPT is mandatory from Armv8.0.

For more information, see:

- Instruction caches.
- Instruction caches.

FEAT_LittleEnd, Support for little-endian at EL1 and above

FEAT_LittleEnd is OPTIONAL from Armv8.0.

If FEAT_LittleEnd is implemented, then FEAT_LittleEndEL0 is implemented.

If FEAT_MixedEnd is implemented, then FEAT_LittleEnd is implemented.

FEAT_LittleEndEL0, Support for little-endian at EL0

FEAT_LittleEndEL0 is OPTIONAL from Armv8.0.

If FEAT_MixedEndEL0 is implemented, then FEAT_LittleEndEL0 is implemented.

FEAT_MixedEnd, Mixed-endian support

FEAT_MixedEnd provides support for mixed-endian configuration.

FEAT_MixedEnd is OPTIONAL from Armv8.0.

If FEAT_MixedEnd is implemented, then FEAT_MixedEndEL0 is implemented.

The following field identifies the presence of FEAT_MixedEnd:

- ID_AA64MMFR0_EL1.BigEnd.

For more information, see:

- Mixed-endian support in AArch64.
- Mixed-endian support in AArch32.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_MixedEndEL0, Mixed-endian support at EL0

FEAT_MixedEndEL0 provides support for mixed-endian at EL0.

FEAT_MixedEndEL0 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_MixedEndEL0:

- ID_AA64MMFR0_EL1.BigEndEL0.
- ID_AA64MMFR0_EL1.BigEnd.

For more information, see:

- Mixed-endian support in AArch64.
- Mixed-endian support in AArch32.

FEAT_PCSRv8, PC Sample-based Profiling extension

FEAT_PCSRv8 introduces support for PC Sample-based Profiling Extension that provides coarse-grained, non-invasive profiling by an external debugger.

FEAT_PCSRv8 is OPTIONAL from Armv8.0.

If FEAT_PCSRv8 is implemented, then FEAT_PCSRv8p2 is not implemented.

The following fields identify the presence of FEAT_PCSRv8:

- DBGDEVID.PCSample.
- DBGDEVID1.PCSROffset.
- EDDEVID1.PCSROffset.
- EDDEVID.PCSample.

For more information, see About the PC Sample-based Profiling Extension.

FEAT_PMULL, Advanced SIMD PMULL instructions

FEAT_PMULL provides the AES* instructions that support multiplication of 64-bit polynomials, PMULL, PMULL2.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMULL is OPTIONAL from Armv8.0.

If FEAT_PMULL is implemented, then FEAT_AES is implemented.

The following field identifies the presence of FEAT_PMULL:

- ID_AA64ISAR0_EL1.AES.

FEAT_PMUv3, PMU extension version 3

The Performance Monitors Extension, FEAT_PMUv3, is an optional non-invasive debug component.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_PMUv3:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see The Performance Monitors Extension.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PMUv3_EXT, External interface to the Performance Monitors

FEAT_PMUv3_EXT indicates support for external access to the FEAT_PMUv3 and/or FEAT_PCSRv8p2 registers.

FEAT_PMUv3_EXT is OPTIONAL from Armv8.0.

If FEAT_PMUv3_EXT is implemented, then FEAT_PMUv3 or FEAT_PCSRv8p2 is implemented.

If FEAT_PMUv3_EXT is implemented, then FEAT_PMUv3_EXT32 or FEAT_PMUv3_EXT64 is implemented.

FEAT_PMUv3_EXT32, 32-bit external interface to the Performance Monitors

FEAT_PMUv3_EXT32 indicates the external Performance Monitors and CoreSight registers are implemented as mostly 32-bit registers.

FEAT_PMUv3_EXT32 is OPTIONAL from Armv8.0.

If FEAT_PMUv3_EXT32 is implemented, then FEAT_PMUv3_EXT is implemented.

If FEAT_PMUv3_EXT32 is implemented, then FEAT_PMUv3_EXT64 is not implemented.

The following field identifies the presence of FEAT_PMUv3_EXT32:

- PMDEVARCH.ARCHPART.

For more information, see Recommended External Interface to the Performance Monitors.

FEAT_SHA1, Advanced SIMD SHA1 instructions

FEAT_SHA1 implements the SHA1* instructions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SHA1 is OPTIONAL from Armv8.0.

If FEAT_SHA1 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SHA1:

- ID_AA64ISAR0_EL1.SHA1.

FEAT_SHA256, Advanced SIMD SHA256 instructions

FEAT_SHA256 implements the SHA256* instructions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SHA256 is OPTIONAL from Armv8.0.

If FEAT_SHA256 is implemented, then FEAT_SHA1 is implemented.

The following field identifies the presence of FEAT_SHA256:

- ID_AA64ISAR0_EL1.SHA2.

FEAT_Secure, Support for Secure state

FEAT_Secure adds support for Secure state.

This feature is not supported if EL2 is using AArch32.

FEAT_Secure is OPTIONAL from Armv8.0.

If FEAT_RME is not implemented and FEAT_EL3 is implemented, then FEAT_Secure is implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SpecSEI, SError interrupt exceptions from speculative reads of memory

Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches.

FEAT_SpecSEI is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_SpecSEI:

- ID_AA64MMFR1_EL1.SpecSEI.
- ID_MMFR4_EL1.SpecSEI.
- ID_MMFR4.SpecSEI.

FEAT_TGran16K, Support for 16KB memory translation granule size at stage 1

FEAT_TGran16K is OPTIONAL from Armv8.0.

When FEAT_AA64EL2 and FEAT_TGran16K are implemented, FEAT_S2TGran16K is implemented.

The following field identifies the presence of FEAT_TGran16K:

- ID_AA64MMFR0_EL1.TGran16.

FEAT_TGran4K, Support for 4KB memory translation granule size at stage 1

FEAT_TGran4K is OPTIONAL from Armv8.0.

When FEAT_AA64EL2 and FEAT_TGran4K are implemented, FEAT_S2TGran4K is implemented.

The following field identifies the presence of FEAT_TGran4K:

- ID_AA64MMFR0_EL1.TGran4.

FEAT_TGran64K, Support for 64KB memory translation granule size at stage 1

FEAT_TGran64K is OPTIONAL from Armv8.0.

When FEAT_AA64EL2 and FEAT_TGran64K are implemented, FEAT_S2TGran64K is implemented.

The following field identifies the presence of FEAT_TGran64K:

- ID_AA64MMFR0_EL1.TGran64.

FEAT_TRC_EXT, Trace external registers

FEAT_TRC_EXT indicates support for external access to the ETMv4 or ETE registers.

FEAT_TRC_EXT is OPTIONAL from Armv8.0.

If FEAT_TRC_EXT is implemented, then FEAT_ETMv4 or FEAT_ETE is implemented.

FEAT_TRC_SR, Trace System registers

FEAT_TRC_SR indicates support for System registers for ETMv4 or ETE.

This feature is supported in both AArch64 and AArch32, but only when either of these are supported at EL1.

FEAT_TRC_SR is OPTIONAL from Armv8.0.

If FEAT_TRC_SR is implemented, then FEAT_ETMv4 or FEAT_ETE is implemented.

The following fields identify the presence of FEAT_TRC_SR:

- ID_AA64DFR0_EL1.TraceVer.
- EDDFR.TraceVer.
- ID_DFR0.CopTrc.
- ID_DFR0_EL1.CopTrc.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_nTLBPA, Intermediate caching of translation table walks

FEAT_nTLBPA introduces a mechanism to identify if the intermediate caching of translation table walks does not include non-coherent caches of previous valid translation table entries since the last completed TLBI applicable to the PE.

This feature is supported in both AArch64 and AArch32 states.

FEAT_nTLBPA is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_nTLBPA:

- ID_AA64MMFR1_EL1.nTLBPA.
- ID_MMFR5_EL1.nTLBPA.
- ID_MMFR5.nTLBPA.

For more information, see:

- TLB maintenance.
- General TLB maintenance requirements.

A2.2.1.1 Features added to the Armv8.0 extension in later releases

- FEAT_CHK.
- FEAT_CLRBHB.
- FEAT_CP15SDISABLE2.
- FEAT_CSV3.
- FEAT_DGH.
- FEAT_ECBHB.
- FEAT_ETS3.
- FEAT_GTG.
- FEAT_PRFMSLC.
- FEAT_RAS.
- FEAT_RPRFM.
- FEAT_SB.
- FEAT_SCTLR2.
- FEAT_SPECRES2.
- FEAT_SPECRES.
- FEAT_SSBS2.
- FEAT_SSBS.
- FEAT_TCR2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.2.2 The Armv8.1 architecture extension

The Armv8.1 architecture extension is an extension to Armv8.0. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.1 compliant if all of the following apply:

- It is Armv8.0 compliant.
- It includes all of the Armv8.1 architectural features that are mandatory.

An Armv8.1 compliant implementation can additionally include:

- Armv8.1 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.2, subject only to those constraints that require that certain features be implemented together.

FEAT_Debugv8p1, Debug v8.1

FEAT_Debugv8p1 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_Debugv8p1:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

FEAT_E2H0, Programming of HCR_EL2.E2H

When FEAT_VHE is implemented, FEAT_E2H0 indicates that HCR_EL2.E2H can be programmed to the value 0. The absence of FEAT_E2H0 results in a relaxed behavior where HCR_EL2.E2H is RES1, with EL2 host mode always enabled.

This feature is supported in AArch64 state only.

FEAT_E2H0 is OPTIONAL from Armv8.0.

If FEAT_E2H0 is implemented, then FEAT_VHE is implemented.

The following field identifies the presence of FEAT_E2H0:

- ID_AA64MMFR4_EL1.E2H0.

FEAT_HAFDBS, Hardware management of the Access flag and dirty state

In Armv8.0, all updates to the translation tables are performed by software. From Armv8.1, for the VMSAv8-64 translation regimes only, hardware can perform updates to the translation tables in two contexts:

- Hardware management of the Access flag.
- Hardware management of dirty state, with updates to a dirty state in the translation tables.

The dirty state is introduced in Armv8.1.

Hardware management of dirty state can be enabled only when hardware management of the Access flag is also enabled.

This feature is supported in AArch64 state only.

FEAT_HAFDBS is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_HAFDBS:

- ID_AA64MMFR1_EL1.HAFDBS.

For more information, see:

- Hardware management of the Access flag.
- Hardware management of the dirty state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_HPDS, Hierarchical permission disables in translations tables

FEAT_HPDS introduces the facility to disable the hierarchical permissions, APTable, PXNTable, and UXNTable, in the translation tables. This disable has no effect on the NSTable bit.

FEAT_HPDS is OPTIONAL from Armv8.0.

FEAT_HPDS is mandatory from Armv8.1.

The following field identifies the presence of FEAT_HPDS:

- ID_AA64MMFR1_EL1.HPDS.

This feature is added only to the VMSAv8-64 translation regimes. Armv8.2 extends this to the AArch32 translation regimes, see FEAT_AA32HPD.

FEAT_LOR, Limited ordering regions

Limited ordering regions allow large systems to perform special Load-Acquire and Store-Release instructions that provide order between the memory accesses to a region of the PA map as observed by a limited set of observers.

This feature is supported in AArch64 state only.

FEAT_LOR is OPTIONAL from Armv8.0.

FEAT_LOR is mandatory from Armv8.1.

The following field identifies the presence of FEAT_LOR:

- ID_AA64MMFR1_EL1.LO.

For more information, see Limited ordering regions.

FEAT_LSE, Large System Extensions

FEAT_LSE introduces a set of atomic instructions:

- Compare and Swap instructions, `CAS` and `CASP`.
- Atomic memory operation instructions, LD<OP> and ST<OP>, where <OP> is one of ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, and UMIN.
- Swap instruction, SWP.

This feature is supported in AArch64 state only.

FEAT_LSE is OPTIONAL from Armv8.0.

FEAT_LSE is mandatory from Armv8.1.

The following field identifies the presence of FEAT_LSE:

- ID_AA64ISAR0_EL1.Atomic.

For more information, see:

- ‘Atomic integer memory operations’.
- ‘Swap operations’.
- ‘Compare and Swap operations’.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PAN, Privileged access never

FEAT_PAN introduces a bit to PSTATE. When the value of this PAN state bit is 1, any privileged data access from EL1, or EL2 when the Effective value of HCR_EL2.E2H is 1, to a virtual memory address that is accessible to data accesses at EL0, generates a Permission fault.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PAN is OPTIONAL from Armv8.0.

FEAT_PAN is mandatory from Armv8.1.

The following fields identify the presence of FEAT_PAN:

- ID_AA64MMFR1_EL1.PAN.
- ID_MMFR3_EL1.PAN.
- ID_MMFR3.PAN.

For more information, see:

- PSTATE.PAN.
- About the PAN bit.

FEAT_PMUv3p1, Armv8.1 PMU extensions

FEAT_PMUv3p1 introduces the following:

- The event number space is extended to 16 bits.
- The HPMD bit is added to MDCR_EL2. This bit disables event counting at EL2.
- The STALL_FRONTEND and STALL_BACKEND events are required to be implemented.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p1 is OPTIONAL from Armv8.0.

In an Armv8.1 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p1 is implemented.

If FEAT_PMUv3p1 is implemented, then FEAT_PMUv3 is implemented.

The following fields identify the presence of FEAT_PMUv3p1:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see ‘Required events’.

FEAT_RDM, Advanced SIMD rounding double multiply-accumulate instructions

FEAT_RDM introduces Rounding Double Multiply Add/Subtract Advanced SIMD instructions. For more information, see:

For the A64 instruction set

- SQRDMLAH (by element).
- SQRDMLAH (vectors).
- SQRDMLSH (by element).
- SQRDMLSH (vector).

For the T32 and A32 instruction sets

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- VQRDMLAH.
- VQRDMLSH.

This feature is supported in both AArch64 and AArch32 states.

FEAT_RDM is OPTIONAL from Armv8.0.

In an Armv8.1 implementation, if FEAT_AdvSIMD is implemented, FEAT_RDM is implemented.

The following fields identify the presence of FEAT_RDM:

- ID_AA64ISAR0_EL1.RDM.
- ID_ISAR5_EL1.RDM.
- ID_ISAR5.RDM.

FEAT_VHE, Virtualization Host Extensions

Armv8.1 introduces the Virtualization Host Extensions (VHE) that provide enhanced support for Type 2 hypervisors in Non-secure state.

FEAT_VHE is OPTIONAL from Armv8.0.

In an Armv8.1 implementation, if FEAT_AA64EL2 is implemented, FEAT_VHE is implemented.

If FEAT_VHE is implemented, then FEAT_LSE, FEAT_Debugv8p1, and FEAT_AA64EL2 are implemented.

The following field identifies the presence of FEAT_VHE:

- ID_AA64MMFR1_EL1.VH.

For more information, see Virtualization Host Extensions

FEAT_VMID16, 16-bit VMID

In an Armv8.1 implementation, when EL2 is using AArch64, the virtual machine identifier (VMID) size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

When implemented, this feature is supported only when EL2 is using AArch64.

FEAT_VMID16 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_VMID16:

- ID_AA64MMFR1_EL1.VMIDBits.

For more information, see VMID size.

A2.2.2.1 Features added to the Armv8.1 extension in later releases

- FEAT_DPB2.
- FEAT_DotProd.
- FEAT_FHM.
- FEAT_FlagM.
- FEAT_PAN3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.2.3 The Armv8.2 architecture extension

The Armv8.2 architecture extension is an extension to Armv8.1. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.2 compliant if all of the following apply:

- It is Armv8.1 compliant.
- It includes all of the Armv8.2 architectural features that are mandatory.

An Armv8.2 compliant implementation can additionally include:

- Armv8.2 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.3, subject only to those constraints that require that certain features be implemented together.

FEAT_AA32HPD, AArch32 Hierarchical permission disables

FEAT_HPDS introduced the ability to disable the hierarchical permissions, APTable, PXNTable, and UXNTable, in the VMSAv8-64 translation regimes. FEAT_AA32HPD extends this functionality to the VMSAv8-32 translation regimes when those regimes are using the Long descriptor Translation Table format.

This feature is supported in AArch32 state only.

FEAT_AA32HPD is OPTIONAL from Armv8.1.

If FEAT_AA32HPD is implemented, then FEAT_AA32 is implemented.

The following fields identify the presence of FEAT_AA32HPD:

- ID_MMFR4_EL1.HPDS.
- ID_MMFR4.HPDS.

For more information, see Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors

FEAT_AA32I8MM, AArch32 Int8 matrix multiply-accumulate

FEAT_AA32I8MM introduces integer matrix multiply-accumulate instructions and dot product instructions.

This feature is supported in AArch32 state only.

FEAT_AA32I8MM is OPTIONAL from Armv8.1.

If FEAT_AA32I8MM is implemented, then FEAT_I8MM is implemented.

If FEAT_AA32I8MM is implemented, then FEAT_AA32 is implemented.

The following fields identify the presence of FEAT_AA32I8MM:

- ID_ISAR6_EL1.I8MM.
- ID_ISAR6.I8MM.

For more information, see:

- Advanced SIMD dot product instructions.
- Advanced SIMD matrix multiply-accumulate instructions.

FEAT_ASMv8p2, Armv8.2 changes to the A64 ISA

FEAT_ASMv8p2 introduces the BFC instruction to the A64 instruction set as an alias of BFM. It also requires that the BFC instruction and the A64 pseudo-instruction REV64 are implemented by assemblers.

In Armv8.0 and Armv8.1, the A64 pseudo-instruction REV64 is OPTIONAL.

FEAT_ASMv8p2 is OPTIONAL from Armv8.1.

FEAT_ASMv8p2 is mandatory from Armv8.2.

For more information, see:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- BFC.
- REV64.

FEAT_DPB, DC CVAP instruction

FEAT_DPB introduces a mechanism to identify and manage persistent memory locations in a shared memory hierarchy, including adding the DC CVAP instruction.

This feature is supported in AArch64 state only.

FEAT_DPB is OPTIONAL from Armv8.1.

FEAT_DPB is mandatory from Armv8.2.

The following field identifies the presence of FEAT_DPB:

- ID_AA64ISAR1_EL1.DPB.

For more information, see Memory hierarchy.

FEAT_Debugv8p2, Debug v8.2

FEAT_Debugv8p2 provides the following changes:

- If the Core power domain is powered up and DoubleLockStatus() == TRUE, EDPRSR.{DLK,SPD,PU} is only permitted to read {UNKNOWN, 0, 0}.
- The definition of Exception Catch debug events is extended to include reset entry.
- All CONSTRAINED UNPREDICTABLE cases that generate Exception Catch debug events are removed.
- Controls are added to EDECCR to control Exception Catch debug event generation on exception return.
- All IMPLEMENTATION DEFINED control of external debug accesses to OSLAR_EL1 is removed.
- ExternalSecureNoninvasiveDebugEnabled() cannot override software controls of counting attributable events in Secure state.

FEAT_Debugv8p2 is OPTIONAL from Armv8.1.

FEAT_Debugv8p2 is mandatory from Armv8.2.

If FEAT_Debugv8p2 is implemented, then FEAT_Debugv8p1 is implemented.

The following fields identify the presence of FEAT_Debugv8p2:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- DBGDIDR.Version.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

For more information, see:

- Exception Catch debug event.
- EDPRSR.{DLK, SPD, PU} and the Core power domain.
- Interaction with EL3.
- External access disabled.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_F32MM, Single-precision floating-point matrix multiply-accumulate

FEAT_F32MM introduces support for the SVE single-precision floating-point matrix multiply-accumulate variant of the FMMLA instruction.

This feature is supported in AArch64 state only.

FEAT_F32MM is OPTIONAL from Armv8.2.

If FEAT_F32MM is implemented, then FEAT_SVE is implemented.

The following field identifies the presence of FEAT_F32MM:

- ID_AA64ZFR0_EL1.F32MM.

FEAT_F64MM, Double-precision floating-point matrix multiply-accumulate

FEAT_F64MM introduces support for the following SVE instructions:

- FMMLA (double-precision floating-point matrix multiply-accumulate variant).
- LD1ROB (scalar plus immediate).
- LD1ROB (scalar plus scalar).
- LD1ROD (scalar plus immediate).
- LD1ROD (scalar plus scalar).
- LD1ROH (scalar plus immediate).
- LD1ROH (scalar plus scalar).
- LD1ROW (scalar plus immediate).
- LD1ROW (scalar plus scalar).
- TRN1, TRN2 (vectors) (128-bit variant).
- UZP1, UZP2 (vectors) (128-bit variant).
- ZIP1, ZIP2 (vectors) (128-bit variant).

This feature is supported in AArch64 state only.

FEAT_F64MM is OPTIONAL from Armv8.2.

If FEAT_F64MM is implemented, then FEAT_SVE is implemented.

The following field identifies the presence of FEAT_F64MM:

- ID_AA64ZFR0_EL1.F64MM.

FEAT_FP16, Half-precision floating-point data processing

FEAT_FP16 supports:

- Half-precision data-processing instructions for Advanced SIMD and floating-point in both AArch64 and AArch32 states.

The following fields enable flushing of denormalized numbers to zero for half-precision data-processing instructions:

- FPCR.FZ16.
- FPSCR.FZ16.

This feature is supported in both AArch64 and AArch32 states.

FEAT_FP16 is OPTIONAL from Armv8.2.

In an Armv8.4 implementation, if FEAT_FHM is not implemented, then FEAT_FP16 is not implemented.

In an Armv8.2 implementation, if FEAT_SVE or FEAT_FHM is implemented, FEAT_FP16 is implemented.

The following fields identify the presence of FEAT_FP16:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64PFR0_EL1.FP.
- ID_AA64PFR0_EL1.AdvSIMD.
- MVFR1_EL1.FPHP.
- MVFR1_EL1.SIMDHP.
- MVFR1.FPHP.
- MVFR1.SIMDHP.

For more information, see:

- Half-precision floating-point formats.
- Flushing denormalized numbers to zero.
- Modified immediate constants in A64 floating-point instructions.

FEAT_HPDS2, Hierarchical permission disables

FEAT_HPDS2 provides a mechanism to allow operating systems or hypervisors to make up to four bits of Translation Table final-level descriptors available for IMPLEMENTATION DEFINED hardware use.

This feature is supported in both AArch64 and AArch32 states.

FEAT_HPDS2 is OPTIONAL from Armv8.1.

When FEAT_AA32EL1 and FEAT_HPDS2 are implemented, FEAT_AA32HPD is implemented.

If FEAT_HPDS2 is implemented, then FEAT_HPDS is implemented.

The following fields identify the presence of FEAT_HPDS2:

- ID_AA64MMFR1_EL1.HPDS.
- ID_MMFR4_EL1.HPDS.
- ID_MMFR4.HPDS.

For more information, see:

- Page Based Hardware attributes.
- Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors.

FEAT_I8MM, AArch64 Int8 matrix multiply-accumulate

FEAT_I8MM introduces integer matrix multiply-accumulate instructions and dot product instructions.

This feature is supported in AArch64 state only.

FEAT_I8MM is OPTIONAL from Armv8.1.

FEAT_I8MM is mandatory from Armv8.6.

The following fields identify the presence of FEAT_I8MM:

- ID_AA64ISAR1_EL1.I8MM.
- ID_AA64ZFR0_EL1.I8MM.

For more information, see:

- SIMD integer dot product.
- SIMD integer matrix multiply-accumulate.
- SVE Integer dot product.
- SVE Integer matrix multiply operations.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_IESB, Implicit Error Synchronization event

FEAT_IESB introduces control for an implicit error synchronization event at exception entry and return.

The implicit error synchronization events affect the same synchronizable asynchronous events that are synchronized by the ESB instruction. This means that if an implementation has either no sources of SError exceptions, or all SError exceptions are not synchronizable errors, then the controls added by this feature have no effect.

This feature is supported in AArch64 state only.

FEAT_IESB is OPTIONAL from Armv8.1.

If FEAT_IESB is implemented, then FEAT_RAS is implemented.

The following field identifies the presence of FEAT_IESB:

- ID_AA64MMFR2_EL1.IESB.

For more information, see Error synchronization event.

FEAT_LPA, Large PA and IPA support

FEAT_LPA:

- Allows a larger physical address (PA) and intermediate physical address (IPA) space of up to 52 bits when using the 64KB translation granule.
- Allows a level 1 block size where the block covers a 4TB address range for the 64KB translation granule if the implementation support 52 bits of PA.

This feature is supported in AArch64 state only.

FEAT_LPA is OPTIONAL from Armv8.1.

The following field identifies the presence of FEAT_LPA:

- ID_AA64MMFR0_EL1.PARange.

For more information about FEAT_LPA, see:

- Implemented physical address size.
- Output address size configuration.
- Intermediate physical address size configuration
- Translation using the 64KB granule
- Table descriptor format
- Block descriptor and Page descriptor formats

FEAT_LSMAOC, AArch32 Load/Store Multiple instruction atomicity and ordering controls

FEAT_LSMAOC introduces controls that disable legacy behavior of AArch32 load multiple and store multiple instructions, and provide a trap of one aspect of this legacy behavior.

This feature is supported in both AArch64 and AArch32 states.

FEAT_LSMAOC is OPTIONAL from Armv8.1.

If FEAT_LSMAOC is implemented, then FEAT_AA32 is implemented.

The following fields identify the presence of FEAT_LSMAOC:

- ID_AA64MMFR2_EL1.LSM.
- ID_MMFR4_EL1.LSM.
- ID_MMFR4.LSM.

For more information, see the register field descriptions and:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Generation of Alignment faults by load/store multiple accesses to Device memory.
- Multi-register loads and stores that access Device memory.
- Taking an interrupt or other exception during a multiple-register load or store.

FEAT_LVA, Large VA support

FEAT_LVA supports a *virtual address* (VA) space for each translation table base register of up to 52 bits when using any of the following:

- VMSAv9-128 translation format.
- VMSAv8-64 translation format with the 64KB translation granule.

This feature is supported in AArch64 state only.

FEAT_LVA is OPTIONAL from Armv8.1.

The following field identifies the presence of FEAT_LVA:

- ID_AA64MMFR2_EL1.VARange.

For more information about FEAT_LVA, see:

- Supported virtual address ranges.
- Input address size configuration.
- Translation using the 64KB granule.

FEAT_PAN2, AT S1E1R and AT S1E1W instruction variants affected by PSTATE.PAN

FEAT_PAN2 introduces variants of the AArch64 AT S1E1R and AT S1E1W instructions and the AArch32 ATS1CPR and ATS1CPW instructions. These instructions factor in the PSTATE.PAN bit when determining whether or not the location will generate a Permission fault for a privileged access, as is reported in the PAR. For more information, see:

For the AArch64 System instructions:

- AT S1E1RP, Address Translate Stage 1 EL1 Read PAN.
- AT S1E1RP, Address Translate Stage 1 EL1 Write PA.

For the AArch32 System instructions:

- ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN.
- ATS1CPRP, Address Translate Stage 1 Current state PL1 write PAN.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PAN2 is OPTIONAL from Armv8.1.

FEAT_PAN2 is mandatory from Armv8.2.

If FEAT_PAN2 is implemented, then FEAT_PAN is implemented.

The following fields identify the presence of FEAT_PAN2:

- ID_AA64MMFR1_EL1.PAN.
- ID_MMFR3_EL1.PAN.
- ID_MMFR3.PAN.

For more information, see:

- Address translation instructions.
- Address translation instructions.
- ATS1C**, Address translation stage 1, current security state.
- Encoding and availability of the address translation instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PCSRv8p2, PC Sample-based Profiling Extension

FEAT_PCSRv8p2 moves the control and implementation of the OPTIONAL PC Sample-based Profiling Extension from the ED*SR Debug registers to PM*SR registers in the Performance Monitors address space.

FEAT_PCSRv8p2 is OPTIONAL from Armv8.1.

If FEAT_PCSRv8p2 is implemented, then FEAT_PCSRv8 is not implemented.

The following field identifies the presence of FEAT_PCSRv8p2:

- PMDEVID.PCSample.

For more information, see The PC Sample-based Profiling Extension.

FEAT_RAS, Reliability, Availability and Serviceability (RAS) Extension

FEAT_RAS introduces the following:

- The Error Synchronization Barrier (ESB) instruction for the A32, T32, and A64 instruction sets.
- Error records.
- The Error synchronization event.
- Additional syndrome information on External Aborts.

FEAT_RAS is OPTIONAL from Armv8.0.

FEAT_RAS is mandatory from Armv8.2.

The following fields identify the presence of FEAT_RAS:

- ID_AA64PFR0_EL1.RAS.
- ID_PFR0_EL1.RAS.
- ID_PFR0.RAS.

For more information, see:

- Reliability, Availability, and Serviceability.
- RAS PE Architecture.
- *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

FEAT_SHA3, Advanced SIMD SHA3 instructions

FEAT_SHA3 introduces Advanced SIMD instructions that support SHA3 functionality.

These instructions are added to the A64 instruction set only.

FEAT_SHA3 is OPTIONAL from Armv8.1.

If FEAT_SHA3 is implemented, then FEAT_Crypto is implemented.

If FEAT_SHA3 is implemented, then FEAT_SHA256 and FEAT_SHA1 are implemented.

The following field identifies the presence of FEAT_SHA3:

- ID_AA64ISAR0_EL1.SHA3.

For more information, see FEAT_SHA3, SHA3 functionality.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SHA512, Advanced SIMD SHA512 instructions

FEAT_SHA512 introduces Advanced SIMD instructions that support SHA2-512 functionality.

These instructions are added to the A64 instruction set only.

FEAT_SHA512 is OPTIONAL from Armv8.1.

If FEAT_SHA512 is implemented, then FEAT_Crypto is implemented.

If FEAT_SHA512 is implemented, then FEAT_SHA256 and FEAT_SHA1 are implemented.

The following field identifies the presence of FEAT_SHA512:

- ID_AA64ISAR0_EL1.SHA2.

For more information, see FEAT_SHA512, SHA2-512 functionality.

FEAT_SM3, Advanced SIMD SM3 instructions

FEAT_SM3 introduces Advanced SIMD instructions that support the Chinese cryptography algorithm SM3.

These instructions are added to the A64 instruction set only.

FEAT_SM3 is OPTIONAL from Armv8.1.

If FEAT_SM3 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SM3:

- ID_AA64ISAR0_EL1.SM3.

For more information, see FEAT_SM3, SM3 functionality.

FEAT_SM4, Advanced SIMD SM4 instructions

FEAT_SM4 introduces Advanced SIMD instructions that support the Chinese cryptography algorithm SM4.

These instructions are added to the A64 instruction set only.

FEAT_SM4 is OPTIONAL from Armv8.1.

If FEAT_SM4 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SM4:

- ID_AA64ISAR0_EL1.SM4.

For more information, see FEAT_SM4, SM4 functionality.

FEAT_SPE, Statistical Profiling Extension

FEAT_SPE provides a non-invasive method of sampling software and hardware using randomized sampling of either architectural instructions, as defined by the instruction set architecture, or by microarchitectural operations.

This feature is supported in AArch64 state only.

FEAT_SPE is OPTIONAL from Armv8.1.

When FEAT_SPE and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

The following field identifies the presence of FEAT_SPE:

- ID_AA64DFR0_EL1.PMSVer.

For more information, see The Statistical Profiling Extension.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPE_ArchInst, Statistical Profiling architectural instruction sampling

FEAT_SPE_ArchInst provides sampling of architectural instructions by FEAT_SPE. If FEAT_SPE is implemented and FEAT_SPE_ArchInst is not implemented, then the PE samples microarchitectural operations.

This feature is supported in AArch64 state only.

FEAT_SPE_ArchInst is OPTIONAL.

If FEAT_SPE_ArchInst is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPE_ArchInst:

- PMSIDR_EL1.ArchInst.

For more information, see The Statistical Profiling Extension.

FEAT_SPE_ERnd, Statistical Profiling random sampling at end of period

FEAT_SPE_ERnd provides a secondary counter, PMSICR_EL1.ECOUNT, that is used after the primary sample counter, PMSICR_EL1.COUNT, reaches zero when PMSIRR_EL1.RND is 1. If FEAT_SPE_ERnd is not implemented, then bits [7:0] of the primary counter are used to provide a random offset.

This feature is supported in AArch64 state only.

FEAT_SPE_ERnd is OPTIONAL.

If FEAT_SPE_ERnd is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPE_ERnd:

- PMSIDR_EL1.ERnd.

For more information, see The Statistical Profiling Extension.

FEAT_SPE_LDS, Statistical Profiling data source packet generation

FEAT_SPE_LDS indicates that the Statistical Profiling Unit (SPU) can generate Data Source packets.

This feature is supported in AArch64 state only.

FEAT_SPE_LDS is OPTIONAL.

If FEAT_SPE_LDS is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPE_LDS:

- PMSIDR_EL1.LDS.

FEAT_SVE, Scalable Vector Extension

The Scalable Vector Extension includes the following functionality:

- Configurable vector length with scalable vector lengths from 128 bits up to 2048 bits.
- Predication using scalable predicate registers from 16 bits up to 256 bits.
- Instructions that operate on scalable size vectors and predicates.
- Gather-load and scatter-store.
- Software-managed speculative vectorization.
- System registers and fields to configure the Effective SVE vector length and traps.

The Scalable Vector Extension complements the AArch64 Advanced SIMD and floating-point functionality. SVE does not replace the AArch64 Advanced SIMD and floating-point functionality.

This feature is supported in AArch64 state only.

FEAT_SVE is OPTIONAL from Armv8.2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SVE is implemented, then FEAT_FCMA and FEAT_FP16 are implemented.

When FEAT_SVE and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

The following field identifies the presence of FEAT_SVE:

- ID_AA64PFR0_EL1.SVE.

FEAT_TTCNP, Translation table Common not private translations

FEAT_TTCNP permits multiple PEs in the same Inner Shareable domain to use the same translation tables for a given stage of address translation.

This facility is available for all VMSAv8-64 translation regimes and for VMSAv8-32 translation stages that use the Long descriptor Translation Table format.

This feature is supported in both AArch64 and AArch32 states.

FEAT_TTCNP is OPTIONAL from Armv8.1.

FEAT_TTCNP is mandatory from Armv8.2.

The following fields identify the presence of FEAT_TTCNP:

- ID_AA64MMFR2_EL1.CnP.
- ID_MMFR4_EL1.CnP.
- ID_MMFR4.CnP.

For more information, see:

- Common not private translations.
- Common not private translations in VMSAv8-32.

FEAT_UAO, Unprivileged Access Override control

FEAT_UAO introduces the UAO bit to PSTATE. When the Effective value of UAO is 1, if memory accesses that are described as unprivileged are made by the unprivileged memory access instructions, they are privileged.

This feature is supported in AArch64 state only.

FEAT_UAO is OPTIONAL from Armv8.1.

FEAT_UAO is mandatory from Armv8.2.

The following field identifies the presence of FEAT_UAO:

- ID_AA64MMFR2_EL1.UAO.

For more information, see:

- PSTATE.UAO.
- Load/store unprivileged.
- UAO.

FEAT_XNX, Translation table stage 2 Unprivileged Execute-never

FEAT_XNX extends the stage 2 translation table access permissions to provide control of whether memory is executable at EL0 independent of whether it is executable at EL1.

This facility is available for stage 2 translation stages in VMSAv8-64 and VMSAv8-32.

This feature is supported in both AArch64 and AArch32 states.

FEAT_XNX is OPTIONAL from Armv8.1.

In an Armv8.2 implementation, if FEAT_EL2 is implemented, FEAT_XNX is implemented.

The following fields identify the presence of FEAT_XNX:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR1_EL1.XNX.
- ID_MMFR4_EL1.XNX.
- ID_MMFR4.XNX.

For more information, see:

- Stage 2 instruction execution using Direct permissions.
- Access permissions for instruction execution.

A2.2.3.1 Features added to the Armv8.2 extension in later releases

- FEAT_AA32BF16.
- FEAT_BF16.
- FEAT_EBF16.
- FEAT_EVT.
- FEAT_LRCPC2.
- FEAT_LRCPC3.
- FEAT_LSE2.
- FEAT_MPAM.
- FEAT_MPAMv1p0.
- FEAT_PAuth2.
- FEAT_RASSAv1p1.
- FEAT_RASv1p1.

A2.2.4 The Armv8.3 architecture extension

The Armv8.3 architecture extension is an extension to Armv8.2. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.3 compliant if all of the following apply:

- It is Armv8.2 compliant.
- It includes all of the Armv8.3 architectural features that are mandatory.

An Armv8.3 compliant implementation can additionally include:

- Armv8.3 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.4, subject only to those constraints that require that certain features be implemented together.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_CCIDX, Extended cache index

FEAT_CCIDX introduces the following registers to allow caches to be described with greater numbers of sets and greater associativity:

- A 64-bit format of CCSIDR_EL1.
- CCSIDR2_EL1.
- CCSIDR2.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CCIDX is OPTIONAL from Armv8.2.

The following fields identify the presence of FEAT_CCIDX:

- ID_AA64MMFR2_EL1.CCIDX.
- ID_MMFR4_EL1.CCIDX.
- ID_MMFR4.CCIDX.

For more information, see:

- Possible formats of the Cache Size Identification Register, CCSIDR_EL1.
- Possible formats of the Cache Size Identification Register, CCSIDR and CCSIDR2.

FEAT_CONSTPACFIELD, PAC algorithm enhancement

FEAT_CONSTPACFIELD introduces functionality that permits an implementation with pointer authentication to use the value of bit[55] in the virtual address to determine the size of the PAC field when adding a PAC to the virtual address, even when the top byte is not being ignored.

This feature is supported in AArch64 state only.

FEAT_CONSTPACFIELD is OPTIONAL from Armv8.2.

If FEAT_CONSTPACFIELD is implemented, then FEAT_PAuth2 is implemented.

The following field identifies the presence of FEAT_CONSTPACFIELD:

- ID_AA64ISAR2_EL1.PAC_frac.

For more information, see PAC field.

FEAT_DoPD, Debug over Powerdown

FEAT_DoPD provides a debug programmers' model where all debug and PMU registers are in the Core power domain and all CTI registers are in the Debug power domain. Power control is provided by an OPTIONAL external powerup request mechanism.

When FEAT_DoPD is implemented:

- The optional Software Lock is not implemented by the architecturally defined debug components in the PE Core power domain.
- If an ETMv4 trace unit is implemented, the ETM must implement:
  - ETMv4.2 or later.
  - The Unified Power Domain Model.
- If FEAT_ETE is implemented, the trace unit always implements a single power domain.

FEAT_DoPD is OPTIONAL from Armv8.2.

If FEAT_DoPD is implemented, then FEAT_DoubleLock is not implemented.

If FEAT_DoPD is implemented, then FEAT_Debugv8p2 is implemented.

The following field identifies the presence of FEAT_DoPD:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EDDEVID.DebugPower.

For more information, see Debug Reset and Powerdown Support.

FEAT_EPAC, Enhanced pointer authentication

FEAT_EPAC introduces functionality that permits setting the Pointer Authentication Code (PAC) field to 0 on performing a PAC operation on a non-canonical address.

This feature is supported in AArch64 state only.

FEAT_EPAC is OPTIONAL from Armv8.2.

If FEAT_EPAC is implemented, then FEAT_PAuth is implemented.

If FEAT_EPAC is implemented, then FEAT_PAuth2 is not implemented.

The following fields identify the presence of FEAT_EPAC:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

FEAT_FCMA, Floating-point complex number instructions

FEAT_FCMA introduces instructions for floating-point multiplication and addition of complex numbers.

These instructions are added to the A64 and A32/T32 instruction sets.

FEAT_FCMA is OPTIONAL from Armv8.2.

In an Armv8.3 implementation, if FEAT_FP is implemented, FEAT_FCMA is implemented.

If FEAT_FCMA is implemented, then FEAT_FP is implemented.

The following fields identify the presence of FEAT_FCMA:

- ID_AA64ISAR1_EL1.FCMA.
- ID_ISAR5_EL1.VCMA.
- ID_ISAR5.VCMA.

For more information, see:

- AArch64 Advanced SIMD complex number arithmetic instructions.
- AArch32 Advanced SIMD complex number arithmetic instructions.

FEAT_FPAC, Faulting on AUT* instructions

FEAT_FPAC introduces faulting on an AUT* instruction.

FEAT_FPAC is added as a further extension to FEAT_PAuth2.

This feature is supported in AArch64 state only.

FEAT_FPAC is OPTIONAL from Armv8.2.

If FEAT_FPAC is implemented, then FEAT_PAuth2 is implemented.

The following fields identify the presence of FEAT_FPAC:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Faulting on pointer authentication.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FPACCOMBINE, Faulting on combined pointer authentication instructions

FEAT_FPACCOMBINE introduces faulting on the combined instructions that perform pointer authentication.

FEAT_FPACCOMBINE is added as a further extension to FEAT_FPAC.

FEAT_FPACCOMBINE is OPTIONAL from Armv8.2.

If FEAT_FPACCOMBINE is implemented, then FEAT_FPAC is implemented.

The following fields identify the presence of FEAT_FPACCOMBINE:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Faulting on pointer authentication.

FEAT_FPACC_SPEC, Speculative behavior of pointer authentication instructions

FEAT_FPACC_SPEC introduces consistent impact of speculation for instructions that perform authentication.

FEAT_FPACC_SPEC is added as a further extension to FEAT_FPACCOMBINE.

This feature is supported in AArch64 state only.

FEAT_FPACC_SPEC is OPTIONAL from Armv8.2.

If FEAT_FPACC_SPEC is implemented, then FEAT_FPACCOMBINE is implemented.

The following field identifies the presence of FEAT_FPACC_SPEC:

- ID_AA64MMFR3_EL1.Spec_FPACC.

For more information, see Faulting on pointer authentication.

FEAT_JSCVT, JavaScript conversion instructions

FEAT_JSCVT introduces JavaScript convert instructions that truncate a double-precision value to a 32-bit signed integer, setting the condition flags to indicate whether the converted value was in range.

These instructions are added to the A64 and A32/T32 instruction sets.

FEAT_JSCVT is OPTIONAL from Armv8.2.

In an Armv8.3 implementation, if FEAT_FP is implemented, FEAT_JSCVT is implemented.

If FEAT_JSCVT is implemented, then FEAT_FP is implemented.

The following fields identify the presence of FEAT_JSCVT:

- ID_AA64ISAR1_EL1.JSCVT.
- ID_ISAR6_EL1.JSCVT.
- ID_ISAR6.JSCVT.

For more information, see:

- Floating-point conversion.
- About the A64 Advanced SIMD and floating-point instructions.
- Advanced SIMD and floating-point instructions.
- Floating-point data-processing instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LRCPC, Load-Acquire RCpc instructions

FEAT_LRCPC introduces instructions that support the weaker Release Consistency processor consistent (RCpc) model that enables the reordering of a Store-Release followed by a Load-Acquire to a different address.

These instructions are added to the A64 instruction set only.

FEAT_LRCPC is OPTIONAL from Armv8.2.

FEAT_LRCPC is mandatory from Armv8.3.

The following field identifies the presence of FEAT_LRCPC:

- ID_AA64ISAR1_EL1.LRCPC.

For more information, see:

- Load-Acquire, Load-AcquirePC, and Store-Release.
- Load-Acquire/Store-Release.

FEAT_NV, Nested Virtualization

FEAT_NV provides support for a Guest Hypervisor to run in EL1 and ensures that the Guest Hypervisor is unaware that it is running at that Exception level. A Guest Hypervisor is supported regardless of the Effective value of HCR_EL2.E2H.

This feature is supported in AArch64 state only.

FEAT_NV is OPTIONAL from Armv8.2.

If FEAT_NV is implemented, then FEAT_EL2 is implemented.

The following fields identify the presence of FEAT_NV:

- ID_AA64MMFR4_EL1.NV_frac.
- ID_AA64MMFR2_EL1.NV.

For more information, see Nested virtualization.

FEAT_PACIMP, Pointer authentication - IMPLEMENTATION DEFINED algorithm

FEAT_PACIMP permits an IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

This feature is supported in AArch64 state only.

FEAT_PACIMP is OPTIONAL from Armv8.2.

If FEAT_PACIMP is implemented, then FEAT_PAuth is implemented.

If FEAT_PACIMP is implemented, then FEAT_PACQARMA5 is not implemented.

If FEAT_PACIMP is implemented, then FEAT_PACQARMA3 is not implemented.

The following fields identify the presence of FEAT_PACIMP:

- ID_AA64ISAR1_EL1.GPI.
- ID_AA64ISAR1_EL1.API.

For more information, see Pointer authentication.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PACQARMA3, Pointer authentication - QARMA3 algorithm

FEAT_PACQARMA3 introduces the QARMA3 cryptographic algorithm for PAC calculation.

This feature is supported in AArch64 state only.

FEAT_PACQARMA3 is OPTIONAL from Armv8.2.

If FEAT_PACQARMA3 is implemented, then FEAT_PAuth is implemented.

If FEAT_PACQARMA3 is implemented, then FEAT_PACQARMA5 is not implemented.

If FEAT_PACQARMA3 is implemented, then FEAT_PACIMP is not implemented.

The following fields identify the presence of FEAT_PACQARMA3:

- ID_AA64ISAR2_EL1.GPA3.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

FEAT_PACQARMA5, Pointer authentication - QARMA5 algorithm

FEAT_PACQARMA5 introduces the QARMA5 cryptographic algorithm for PAC calculation.

This feature is supported in AArch64 state only.

FEAT_PACQARMA5 is OPTIONAL from Armv8.2.

If FEAT_PACQARMA5 is implemented, then FEAT_PAuth is implemented.

If FEAT_PACQARMA5 is implemented, then FEAT_PACQARMA3 is not implemented.

If FEAT_PACQARMA5 is implemented, then FEAT_PACIMP is not implemented.

The following fields identify the presence of FEAT_PACQARMA5:

- ID_AA64ISAR1_EL1.GPA.
- ID_AA64ISAR1_EL1.APA.

For more information, see Pointer authentication.

FEAT_PAuth, Pointer authentication

FEAT_PAuth introduces functionality that supports address authentication of the contents of a register before that register is used as the target of an indirect branch, or as a load.

When FEAT_PAuth is implemented, all of the following must be true:

- Exactly one of the PAC algorithms is implemented.
- The PACGA instruction and other Pointer authentication instructions use the same algorithm.

The PAC algorithm features are:

- FEAT_PACQARMA5.
- FEAT_PACIMP.
- FEAT_PACQARMA3.

This feature is supported in AArch64 state only.

FEAT_PAuth is OPTIONAL from Armv8.2.

FEAT_PAuth is mandatory from Armv8.3.

If FEAT_PAuth is implemented, then FEAT_PACQARMA5, FEAT_PACIMP, or FEAT_PACQARMA3 is implemented.

For more information, see Pointer authentication.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPEv1p1, Statistical Profiling Extension version 1.1

FEAT_SPEv1p1 introduces an Alignment Flag in the Events packet and filtering on this event using PMSEVFR_EL1, together with support for the profiling of Scalable Vector Extension operations.

This feature is supported in AArch64 state only.

FEAT_SPEv1p1 is OPTIONAL from Armv8.2.

In an Armv8.5 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p1 is implemented.

If FEAT_SPEv1p1 is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPEv1p1:

- ID_AA64DFR0_EL1.PMSVer.

An implementation that includes FEAT_SVE and the Statistical Profiling Extension is strongly recommended to implement FEAT_SPEv1p1 whenever possible.

For more information, see:

- The Statistical Profiling Extension.
- Statistical Profiling Extension Sample Record Specification.

A2.2.5 The Armv8.4 architecture extension

The Armv8.4 architecture extension is an extension to Armv8.3. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.4 compliant if all of the following apply:

- It is Armv8.3 compliant.
- It includes all of the Armv8.4 architectural features that are mandatory.

An Armv8.4 compliant implementation can additionally include:

- Armv8.4 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.5, subject only to those constraints that require that certain features be implemented together.

FEAT_AMU_EXT, External Activity Monitors

FEAT_AMU_EXT indicates support for external access to the Activity Monitors.

FEAT_AMU_EXT is OPTIONAL.

If FEAT_AMU_EXT is implemented, then FEAT_AMU_EXT32 or FEAT_AMU_EXT64 is implemented.

If FEAT_AMU_EXT is implemented, then FEAT_AMUv1 is implemented.

FEAT_AMU_EXT32, AArch32 External Activity Monitors

FEAT_AMU_EXT32 indicates the external AMU registers are implemented as mostly 32-bit registers.

FEAT_AMU_EXT32 is OPTIONAL.

If FEAT_AMU_EXT32 is implemented, then FEAT_AMU_EXT is implemented.

If FEAT_AMU_EXT32 is implemented, then FEAT_AMU_EXT64 is not implemented.

The following field identifies the presence of FEAT_AMU_EXT32:

- AMDEVARCH.ARCHID.

For more information, see Recommended External Interface to the Activity Monitors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AMUv1, Activity Monitors Extension version 1

FEAT_AMUv1 provides a function similar to a subset of the existing Performance Monitors Extension functionality, intended for system management use rather than debugging and profiling.

This feature is supported in both AArch64 and AArch32 states.

FEAT_AMUv1 is OPTIONAL from Armv8.3.

The following fields identify the presence of FEAT_AMUv1:

- ID_AA64PFR0_EL1.AMU.
- ID_PFR0_EL1.AMU.
- ID_PFR0.AMU.
- EDPFR.AMU.

For more information, see The Activity Monitors Extension.

FEAT_BBML1, Translation table break-before-make level 1

FEAT_BBML1 provides support for reducing the requirements for following a break-before-make sequence when changing between table or block sizes for a translation.

This feature is supported in AArch64 state only.

FEAT_BBML1 is OPTIONAL from Armv8.3.

The following field identifies the presence of FEAT_BBML1:

- ID_AA64MMFR2_EL1.BBM.

For more information, see:

- Block descriptor and Page descriptor formats.
- ‘Block translation entry’.
- Support levels for changing block size.

FEAT_BBML2, Translation table break-before-make level 2

FEAT_BBML2 provides support for further reducing the requirements for following a break-before-make sequences when changing between table or block sizes for a translation.

This feature is supported in AArch64 state only.

FEAT_BBML2 is OPTIONAL from Armv8.3.

If FEAT_BBML2 is implemented, then FEAT_BBML1 is implemented.

The following field identifies the presence of FEAT_BBML2:

- ID_AA64MMFR2_EL1.BBM.

For more information, see FEAT_BBML1.

FEAT_CNTSC, Generic Counter Scaling

FEAT_CNTSC introduces a scaling register to the memory-mapped counter module that allows the frequency of the counter that is generated to be scaled from the basic frequency reported in the counter ID mechanisms.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CNTSC is OPTIONAL from Armv8.3.

The following field identifies the presence of FEAT_CNTSC:

- CNTID.CNTSC.

For more information, see CNTCR.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_DIT, Data Independent Timing instructions

FEAT_DIT provides a mechanism for software to indicate that a sequence of code is intended to be independent from data values from certain instructions within the sequence.

This feature is supported in both AArch64 and AArch32 states.

FEAT_DIT is OPTIONAL from Armv8.3.

FEAT_DIT is mandatory from Armv8.4.

The following fields identify the presence of FEAT_DIT:

- ID_AA64PFR0_EL1.DIT.
- ID_PFR0_EL1.DIT.
- ID_PFR0.DIT.

For more information, see:

- About PSTATE.DIT.
- About the DIT bit.

FEAT_Debugv8p4, Debug v8.4

FEAT_Debugv8p4 provides the following mandatory changes:

- The fields MDCR_EL3.{EPMAD, EDAD} control Non-secure access to the debug and PMU registers. The bus Requester is responsible for other debug authentication.
- The Software Lock is obsolete.
- Non-invasive Debug controls are relaxed.
- Secure and Non-secure views of the debug registers are enabled.

FEAT_Debugv8p4 is OPTIONAL from Armv8.3.

FEAT_Debugv8p4 is mandatory from Armv8.4.

If FEAT_Debugv8p4 is implemented, then FEAT_Debugv8p2 is implemented.

If FEAT_SEL2 is implemented, then FEAT_Debugv8p4 is implemented.

The following fields identify the presence of FEAT_Debugv8p4:

- ID_DFR0_EL1.CopDbg.
- ID_AA64DFR0_EL1.DebugVer.
- DBGDIDR.Version.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

For more information, see:

- Definition and constraints of a debugger in the context of external debug.
- External debug interface register access permissions.

FEAT_DotProd, Advanced SIMD dot product instructions

FEAT_DotProd provides instructions to perform a four-way vector dot product of 8-bit integers, accumulating each sum of four products into a 32-bit integer. Each 8-bit input can be treated as a signed or unsigned value.

These instructions are added to the A64 and A32/T32 instruction sets.

FEAT_DotProd is OPTIONAL from Armv8.1.

In an Armv8.4 implementation, if FEAT_AdvSIMD is implemented, FEAT_DotProd is implemented.

The following fields identify the presence of FEAT_DotProd:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64ISAR0_EL1.DP.
- ID_ISAR6_EL1.DP.
- ID_ISAR6.DP.

For more information, see:

- SIMD integer dot product.
- Advanced SIMD dot product instructions.

FEAT_DoubleFault, Double Fault Extension

FEAT_DoubleFault provides controls for routing and masking error exceptions.

This feature is supported in AArch64 state only.

FEAT_DoubleFault is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_AA64EL3 is implemented, FEAT_DoubleFault is implemented.

If FEAT_DoubleFault is implemented, then FEAT_DoubleFault2 or FEAT_AA64EL3 is implemented.

The following field identifies the presence of FEAT_DoubleFault:

- ID_AA64PFR0_EL1.RAS.

For more information, see:

- Taking error exceptions.
- Error synchronization event.

FEAT_FHM, Floating-point half-precision to single-precision multiply-add instructions

FEAT_FHM introduces half-precision to single-precision fused multiply-add instructions.

These instructions are added to the A64 and A32/T32 instruction sets.

FEAT_FHM is OPTIONAL from Armv8.1.

In an Armv8.4 implementation, if FEAT_FP16 is implemented, FEAT_FHM is implemented.

In an Armv8.2 implementation, if FEAT_FHM is implemented, FEAT_FP16 is implemented.

The following fields identify the presence of FEAT_FHM:

- ID_AA64ISAR0_EL1.FHM.
- ID_ISAR6_EL1.FHM.
- ID_ISAR6.FHM.

For more information, see:

- SIMD arithmetic.
- SIMD by element arithmetic.
- Advanced SIMD multiply instructions.

FEAT_FlagM, Condition flag manipulation instructions

FEAT_FlagM provides instructions that manipulate the PSTATE.{N,Z,C,V} flags.

These instructions are added to the A64 instruction set only.

FEAT_FlagM is OPTIONAL from Armv8.1.

FEAT_FlagM is mandatory from Armv8.4.

The following field identifies the presence of FEAT_FlagM:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64ISAR0_EL1.TS.

For more information, see Flag manipulation instructions.

FEAT_IDST, ID space trap handling

FEAT_IDST causes all AArch64 read accesses to the feature ID space when exceptions are generated to be reported in ESR_ELx using the EC code `0x18`.

This feature is supported in AArch64 state only.

FEAT_IDST is OPTIONAL from Armv8.3.

FEAT_IDST is mandatory from Armv8.4.

The following field identifies the presence of FEAT_IDST:

- ID_AA64MMFR2_EL1.IDS.

FEAT_LRCPC2, Load-Acquire RCpc instructions version 2

FEAT_LRCPC2 provides versions of `LDAPR` and `STLR` with a 9-bit unscaled signed immediate offset.

These instructions are added to the A64 instruction set only.

FEAT_LRCPC2 is OPTIONAL from Armv8.2.

FEAT_LRCPC2 is mandatory from Armv8.4.

If FEAT_LRCPC2 is implemented, then FEAT_LRCPC is implemented.

The following field identifies the presence of FEAT_LRCPC2:

- ID_AA64ISAR1_EL1.LRCPC.

For more information, see:

- Changes to single-copy atomicity in Armv8.4.
- Load-Acquire/Store-Release.
- A64 instructions that are changed in Debug state.

FEAT_LSE2, Large System Extensions version 2

FEAT_LSE2 introduces changes to single-copy atomicity requirements for loads and stores, and changes to alignment requirements for loads and stores.

This feature is supported in AArch64 state only.

FEAT_LSE2 is OPTIONAL from Armv8.2.

FEAT_LSE2 is mandatory from Armv8.4.

The following field identifies the presence of FEAT_LSE2:

- ID_AA64MMFR2_EL1.AT.

For more information, see:

- Requirements for single-copy atomicity.
- Alignment of data accesses.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_MPAM, Memory System Resource Partitioning and Monitoring Extension

The MPAM Extension provides a framework for memory-system component controls that partition one or more of the performance resources of the component. There are three versions of the MPAM extension: v1p0, v1p1, and v0p1.

This feature is supported in AArch64 state only.

FEAT_MPAM is OPTIONAL from Armv8.2.

If FEAT_MPAM is implemented, then FEAT_MPAMv1p0 or FEAT_MPAMv0p1 is implemented.

For more information, see MPAM PE Architecture.

FEAT_MPAMv1p0, Memory Partitioning and Monitoring Extension version 1.0

FEAT_MPAMv1p0 introduces support for version 1.0 of the MPAM extension.

This feature is supported in AArch64 state only.

FEAT_MPAMv1p0 is OPTIONAL from Armv8.2.

If FEAT_MPAMv1p0 is implemented, then FEAT_MPAM is implemented.

If FEAT_MPAMv1p0 is implemented, then FEAT_MPAMv0p1 is not implemented.

The following field identifies the presence of FEAT_MPAMv1p0:

- ID_AA64PFR0_EL1.MPAM.

For more information, see MPAM PE Architecture.

FEAT_NV2, Enhanced nested virtualization support

FEAT_NV2 supports nested virtualization by redirecting register accesses that would be trapped to EL1 and EL2 to access memory instead. The address of the memory access depends on information held in VNCR_EL2.

This feature is supported in AArch64 state only.

FEAT_NV2 is OPTIONAL from Armv8.3.

If FEAT_NV2 is implemented, then FEAT_NV is implemented.

The following fields identify the presence of FEAT_NV2:

- ID_AA64MMFR4_EL1.NV_frac.
- ID_AA64MMFR2_EL1.NV.

For more information, see Enhanced support for nested virtualization.

FEAT_PMUv3p4, Arm8.4 PMU extensions

FEAT_PMUv3p4 introduces:

- PMMIR_EL1
- PMMIR registers.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p4 is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p4 is implemented.

If FEAT_PMUv3p4 is implemented, then FEAT_PMUv3p1 is implemented.

The following fields identify the presence of FEAT_PMUv3p4:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see PMU Event Descriptions.

FEAT_RASSAv1p1, RAS version 1.1 System Architecture

FEAT_RASSAv1p1 implements the RAS System Architecture v1.1, including support for the following:

- Additional ERR<n>MISC<m> registers.
- The optional RAS Common Fault Injection Model Extension.

FEAT_RASSAv1p1 is OPTIONAL from Armv8.2.

In an Armv8.4 implementation, if FEAT_RAS is implemented, FEAT_RASSAv1p1 is implemented.

If FEAT_RASv1p1 is implemented, then FEAT_RASSAv1p1 is implemented.

FEAT_RASv1p1, RAS extension v1.1

FEAT_RASv1p1 introduces support for System register access to the following RAS System Architecture v1.1 features:

- Additional ERR<n>MISC<m> registers.
- The optional RAS Common Fault Injection Model Extension.

This feature is supported in both AArch64 and AArch32 states.

FEAT_RASv1p1 is OPTIONAL from Armv8.2.

In an Armv8.4 implementation, if FEAT_RAS is implemented, FEAT_RASv1p1 is implemented.

If FEAT_RASv1p1 is implemented, then FEAT_RAS is implemented.

If FEAT_RASv1p1 is implemented, then FEAT_RASSAv1p1 is implemented.

The following fields identify the presence of FEAT_RASv1p1:

- ID_AA64PFR0_EL1.RAS.
- ID_AA64PFR1_EL1.RAS_frac.
- ID_PFR0_EL1.RAS.
- ID_PFR2_EL1.RAS_frac.
- ID_PFR0.RAS.
- ID_PFR2.RAS_frac.

For more information, see *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

FEAT_S2FWB, Stage 2 forced Write-Back

FEAT_S2FWB reduces the requirement of additional cache maintenance instructions in systems where the data Cacheability attributes used by the Guest operating system are different from those expected by the Hypervisor. If this feature is implemented, there is no meaningful distinction between the Inner and Outer Shareability domains for accesses to Normal Cacheable memory.

This feature is supported in AArch64 state only.

FEAT_S2FWB is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_EL2 is implemented, FEAT_S2FWB is implemented.

The following field identifies the presence of FEAT_S2FWB:

- ID_AA64MMFR2_EL1.FWB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- Block descriptor and Page descriptor formats.
- Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled.

FEAT_SEL2, Secure EL2

FEAT_SEL2 permits EL2 to be implemented in Secure state. When Secure EL2 is enabled, a translation regime is introduced that follows the same format as the other Secure translation regimes.

This feature is not supported if EL2 is using AArch32.

FEAT_SEL2 is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_AA64EL2 and FEAT_Secure are implemented, FEAT_SEL2 is implemented.

If FEAT_SEL2 is implemented, then FEAT_TTST is implemented.

If FEAT_SEL2 is implemented, then FEAT_PCSRv8 is not implemented.

If FEAT_SEL2 is implemented, then FEAT_EL2 is implemented.

If FEAT_SEL2 is implemented, then FEAT_Secure is implemented.

The following field identifies the presence of FEAT_SEL2:

- ID_AA64PFR0_EL1.SEL2.

For more information, see:

- Security states
- Translation regimes.

FEAT_TLBIOS, TLB invalidate instructions in Outer Shareable domain

FEAT_TLBIOS provides TLBI maintenance instructions that extend to the Outer Shareable domain.

This feature is supported in AArch64 state only.

FEAT_TLBIOS is OPTIONAL from Armv8.3.

FEAT_TLBIOS is mandatory from Armv8.4.

The following field identifies the presence of FEAT_TLBIOS:

- ID_AA64ISAR0_EL1.TLB.

For more information, see TLB maintenance instructions.

FEAT_TLBIRANGE, TLB invalidate range instructions

FEAT_TLBIRANGE provides TLBI maintenance instructions that apply to a range of input addresses.

This feature is supported in AArch64 state only.

FEAT_TLBIRANGE is OPTIONAL from Armv8.3.

FEAT_TLBIRANGE is mandatory from Armv8.4.

If FEAT_TLBIRANGE is implemented, then FEAT_TLBIOS is implemented.

The following field identifies the presence of FEAT_TLBIRANGE:

- ID_AA64ISAR0_EL1.TLB.

For more information, see:

- TLB maintenance instructions.
- TLB maintenance instructions that do not apply to a range of addresses.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TRF, Self-hosted Trace extensions

FEAT_TRF introduces control of trace in a self-hosted system through System registers.

The feature provides:

- Control of Exception levels and Security states where trace generation is prohibited.
- Control of whether an offset is used for the timestamp recorded with trace information.
- A context synchronization instruction, TSB CSYNC, that can be used to prevent reordering of trace operation accesses with respect to other accesses of the same System registers.

This feature is supported in both AArch64 and AArch32 states.

FEAT_TRF is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_ETMv4 is implemented, FEAT_TRF is implemented.

If FEAT_TRF is implemented, then FEAT_TRC_SR is implemented.

The following fields identify the presence of FEAT_TRF:

- ID_DFR0_EL1.TraceFilt.
- ID_AA64DFR0_EL1.TraceFilt.
- ID_DFR0.TraceFilt.
- EDDFR.TraceFilt.

For more information on FEAT_TRF, see:

- AArch64 Self-hosted Trace.
- AArch32 Self-hosted Trace.

FEAT_TTL, Translation Table Level

FEAT_TTL provides the TTL field to indicate the level of translation table walk holding the leaf entry for the address that is being invalidated. This field is provided in all TLB maintenance instructions that take a VA or an IPA argument.

This feature is supported in AArch64 state only.

FEAT_TTL is OPTIONAL from Armv8.3.

FEAT_TTL is mandatory from Armv8.4.

The following field identifies the presence of FEAT_TTL:

- ID_AA64MMFR2_EL1.TTL.

For more information, see:

- TLB maintenance instructions.
- TLB maintenance instructions that do not apply to a range of addresses.

FEAT_TTST, Small translation tables

FEAT_TTST relaxes the lower limit on the size of translation tables, by increasing the maximum permitted value of the T1SZ and T0SZ fields in TCR_EL1, TCR_EL2, TCR_EL3, VTCR_EL2 and VSTCR_EL2.

This feature is supported in AArch64 state only.

FEAT_TTST is OPTIONAL from Armv8.3.

If FEAT_SEL2 is implemented, then FEAT_TTST is implemented.

The following field identifies the presence of FEAT_TTST:

- ID_AA64MMFR2_EL1.ST.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- Input address size configuration.
- Translation using the 4KB granule.
- Translation using the 16KB granule.
- Translation using the 64KB granule.

A2.2.5.1 Features added to the Armv8.4 extension in later releases

- FEAT_AMU_EXTACR.
- FEAT_NV2p1.

A2.2.6 The Armv8.5 architecture extension

The Armv8.5 architecture extension is an extension to Armv8.4. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.5 compliant if all of the following apply:

- It is Armv8.4 compliant.
- It includes all of the Armv8.5 architectural features that are mandatory.

An Armv8.5 compliant implementation can additionally include:

- Armv8.5 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.6, subject only to those constraints that require that certain features be implemented together.

FEAT_BTI, Branch Target Identification

FEAT_BTI allows memory pages to be guarded against the execution of instructions that are not the intended target of a branch. To do this, it introduces:

- The GP field, which denotes the blocks and pages in stage 1 translation tables that are guarded pages.
- The PSTATE.BTYPE field, which is used to determine whether an access to a guarded memory region will generate a Branch Target exception.
- The BTI instruction, which is used to guard against the execution of instructions that are not the intended target of a branch.

This feature is supported in AArch64 state only.

FEAT_BTI is OPTIONAL from Armv8.4.

FEAT_BTI is mandatory from Armv8.5.

The following field identifies the presence of FEAT_BTI:

- ID_AA64PFR1_EL1.BT.

For more information, see:

- Table descriptor format.
- PSTATE.BTYPE.
- Effect of entering Debug state on PSTATE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_CSV3, Cache Speculation Variant 3

FEAT_CSV3 introduces a mechanism to identify whether data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, can be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CSV3 is OPTIONAL from Armv8.0.

FEAT_CSV3 is mandatory from Armv8.5.

The following field identifies the presence of FEAT_CSV3:

- ID_AA64PFR0_EL1.CSV3.

FEAT_DPB2, DC CVADP instruction

FEAT_DPB2 allows two levels of cache clean to the Point of Persistence by:

- Redefining Point of Persistence, which changes the scope of DC CVAP.
- Defining a Point of Deep Persistence.
- Adding the DC CVADP System instruction.

This feature is supported in AArch64 state only.

FEAT_DPB2 is OPTIONAL from Armv8.1.

FEAT_DPB2 is mandatory from Armv8.5.

If FEAT_DPB2 is implemented, then FEAT_DPB is implemented.

The following field identifies the presence of FEAT_DPB2:

- ID_AA64ISAR1_EL1.DPB.

For more information, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

FEAT_E0PD, Preventing EL0 access to halves of address maps

FEAT_E0PD prevents access at EL0 to half of the addresses in the memory map.

This feature is supported in AArch64 state only. When EL1 is using AArch64 state, this feature affects access to EL0, in either Execution state.

FEAT_E0PD is OPTIONAL from Armv8.4.

FEAT_E0PD is mandatory from Armv8.5.

If FEAT_E0PD is implemented, then FEAT_CSV3 is implemented.

The following field identifies the presence of FEAT_E0PD:

- ID_AA64MMFR2_EL1.E0PD.

For more information, see Preventing EL0 access to halves of the address map.

FEAT_EVT, Enhanced Virtualization Traps

FEAT_EVT introduces additional traps for EL1 and EL0 Cache controls in:

- HCR_EL2
- HCR2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

These traps are independent of existing controls.

This feature is supported in both AArch64 and AArch32 states.

FEAT_EVT is OPTIONAL from Armv8.2.

In an Armv8.5 implementation, if FEAT_EL2 is implemented, FEAT_EVT is implemented.

The following fields identify the presence of FEAT_EVT:

- ID_AA64MMFR2_EL1.EVT.
- ID_MMFR4_EL1.EVT.
- ID_MMFR4.EVT.

FEAT_ExS, Context synchronization and exception handling

FEAT_ExS provides a mechanism to control whether exception entry and exception return are Context Synchronization events.

Fields in the SCTLR_ELx registers enable and disable context synchronization at exception entry and return at an Exception level.

This feature is supported in AArch64 state only.

FEAT_ExS is OPTIONAL from Armv8.4.

The following field identifies the presence of FEAT_ExS:

- ID_AA64MMFR0_EL1.ExS.

For more information, see ‘Context Synchronization event’.

FEAT_FRINTTS, Floating-point to integer instructions

FEAT_FRINTTS provides instructions that round a floating-point number to an integral valued floating-point number that fits in a 32-bit or 64-bit integer number range.

These instructions are added to the A64 instruction set only.

FEAT_FRINTTS is OPTIONAL from Armv8.4.

In an Armv8.5 implementation, if FEAT_FP is implemented, FEAT_FRINTTS is implemented.

If FEAT_FRINTTS is implemented, then FEAT_FP and FEAT_AdvSIMD are implemented.

The following field identifies the presence of FEAT_FRINTTS:

- ID_AA64ISAR1_EL1.FRINTTS.

For more information, see Floating-point round to integral value.

FEAT_FlagM2, Enhancements to flag manipulation instructions

FEAT_FlagM2 provides instructions that convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format described in Condition flags and related instructions.

These instructions are added to the A64 instruction set only.

FEAT_FlagM2 is OPTIONAL from Armv8.4.

If FEAT_FlagM2 is implemented, then FEAT_FlagM is implemented.

The following field identifies the presence of FEAT_FlagM2:

- ID_AA64ISAR0_EL1.TS.

For more information, see Flag manipulation instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_GTG, Guest translation granule size

FEAT_GTG allows a hypervisor to support different granule sizes for stage 2 and stage 1 translation, and allows a nested hypervisor to determine what stage 2 granule sizes are available.

This feature is supported in AArch64 state only.

FEAT_GTG is OPTIONAL from Armv8.0.

In an Armv8.5 implementation, if FEAT_AA64EL2 is implemented, FEAT_GTG is implemented.

The following fields identify the presence of FEAT_GTG:

- ID_AA64MMFR0_EL1.TGran4_2.
- ID_AA64MMFR0_EL1.TGran16_2.
- ID_AA64MMFR0_EL1.TGran64_2.

For more information, see Translation granules.

FEAT_MTE, Memory Tagging Extension

FEAT_MTE provides architectural support for runtime, always-on detection of various classes of memory error to aid with software debugging to eliminate vulnerabilities arising from memory-unsafe languages.

These features are supported in AArch64 state only.

FEAT_MTE is OPTIONAL from Armv8.4.

The following field identifies the presence of FEAT_MTE:

- ID_AA64PFR1_EL1.MTE.

For more information, see:

- The Memory Tagging Extension.
- The AArch64 Application Level Memory Model.
- PMU Event Descriptions.
- The Statistical Profiling Extension.
- Debug State.

FEAT_MTE2, Memory Tagging Extension

FEAT_MTE2 provides architectural support for runtime, always-on detection of various classes of memory error to aid with software debugging to eliminate vulnerabilities arising from memory-unsafe languages.

These features are supported in AArch64 state only.

FEAT_MTE2 is OPTIONAL from Armv8.4.

If FEAT_MTE2 is implemented, then FEAT_MTE is implemented.

The following field identifies the presence of FEAT_MTE2:

- ID_AA64PFR1_EL1.MTE.

For more information, see:

- The Memory Tagging Extension.
- The AArch64 Application Level Memory Model.
- PMU Event Descriptions.
- The Statistical Profiling Extension.
- Debug State.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PMUv3p5, Arm8.5 PMU extensions

FEAT_PMUv3p5 extends event counters to 64-bit event counters, and introduces mechanisms to disable the cycle counter in Secure state, in EL3, and in EL2.

FEAT_PMUv3p5 relaxes the behavior of PMCR.{IMP, IDCODE}, and deprecates use of these fields.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p5 is OPTIONAL from Armv8.4.

In an Armv8.5 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p5 is implemented.

If FEAT_PMUv3p5 is implemented, then FEAT_PMUv3p4 is implemented.

The following fields identify the presence of FEAT_PMUv3p5:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:

- Behavior on overflow.
- Controlling the PMU counters.
- PMU Event Descriptions.

FEAT_RNG, Random number generator

FEAT_RNG introduces the RNDR and RNDRRS registers. Reads to these registers return a 64-bit random number. A read to RNDRRS will cause a reseeding of the random number before the generation of the random number that is returned.

This feature is supported in AArch64 state only.

FEAT_RNG is OPTIONAL from Armv8.4.

The following field identifies the presence of FEAT_RNG:

- ID_AA64ISAR0_EL1.RNDR.

For more information, see:

- Effect of random number generation instructions on Condition flags
- Appendix K14 Random Number Generation

FEAT_RNG_TRAP, Trapping support for RNDR/RNDRRS

FEAT_RNG_TRAP introduces support for EL3 trapping of reads of the RNDR and RNDRRS registers.

This feature is supported in AArch64 state only.

FEAT_RNG_TRAP is OPTIONAL from Armv8.4.

The following field identifies the presence of FEAT_RNG_TRAP:

- ID_AA64PFR1_EL1.RNDR_trap.

FEAT_S2TGran16K, Support for 16KB memory translation granule size at stage 2

FEAT_S2TGran16K is OPTIONAL.

If FEAT_S2TGran16K is implemented, then FEAT_AA64EL2 and FEAT_TGran16K are implemented.

The following field identifies the presence of FEAT_S2TGran16K:

- ID_AA64MMFR0_EL1.TGran16_2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_S2TGran4K, Support for 4KB memory translation granule size at stage 2

FEAT_S2TGran4K is OPTIONAL.

If FEAT_S2TGran4K is implemented, then FEAT_AA64EL2 and FEAT_TGran4K are implemented.

The following field identifies the presence of FEAT_S2TGran4K:

- ID_AA64MMFR0_EL1.TGran4_2.

FEAT_S2TGran64K, Support for 64KB memory translation granule size at stage 2

FEAT_S2TGran64K is OPTIONAL.

If FEAT_S2TGran64K is implemented, then FEAT_AA64EL2 and FEAT_TGran64K are implemented.

The following field identifies the presence of FEAT_S2TGran64K:

- ID_AA64MMFR0_EL1.TGran64_2.

FEAT_SB, Speculation Barrier

Speculation Barrier FEAT_SB introduces a barrier to control speculation.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SB is OPTIONAL from Armv8.0.

FEAT_SB is mandatory from Armv8.5.

The following fields identify the presence of FEAT_SB:

- ID_AA64ISAR1_EL1.SB.
- ID_ISAR6_EL1.SB.
- ID_ISAR6.SB.

For more information, see:

- Speculation Barrier (SB).
- Barriers and CLREX instructions.
- Speculation Barrier (SB).
- Miscellaneous instructions.

FEAT_SPECRES, Speculation restriction instructions

FEAT_SPECRES introduces System instructions that prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later speculative execution within that context, to the extent that the speculative execution is observable through side channels.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SPECRES is OPTIONAL from Armv8.0.

FEAT_SPECRES is mandatory from Armv8.5.

The following fields identify the presence of FEAT_SPECRES:

- ID_AA64ISAR1_EL1.SPECRES.
- ID_ISAR6_EL1.SPECRES.
- ID_ISAR6.SPECRES.

For more information, see:

- Prediction restriction instructions.
- Execution, data prediction and prefetching restriction System instructions.
- Execution and data prediction restriction System instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SSBS, Speculative Store Bypass Safe

FEAT_SSBS allows software to indicate whether hardware is permitted to load or store speculatively in a manner that could give rise to a cache timing side channel, which in turn could be used to derive an address from values loaded to a register from memory.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SSBS is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_SSBS:

- ID_AA64PFR1_EL1.SSBS.
- ID_PFR2_EL1.SSBS.
- ID_PFR2.SSBS.

For more information, see:

- AArch64 Speculative Store Bypass Safe.
- AArch32 Speculative Store Bypass Safe.

FEAT_SSBS2, MRS and MSR instructions for SSBS version 2

FEAT_SSBS2 provides controls for the MSR and MRS instructions to read and write the PSTATE.SSBS field.

This feature is supported in AArch64 state only.

FEAT_SSBS2 is OPTIONAL from Armv8.0.

If FEAT_SSBS2 is implemented, then FEAT_SSBS is implemented.

The following field identifies the presence of FEAT_SSBS2:

- ID_AA64PFR1_EL1.SSBS.

For more information, see:

- AArch64 Speculative Store Bypass Safe.
- AArch32 Speculative Store Bypass Safe.

A2.2.6.1 Features added to the Armv8.5 extension in later releases

- FEAT_MTE3.
- FEAT_MTE_ASYNC.

A2.2.7 The Armv8.6 architecture extension

The Armv8.6 architecture extension is an extension to Armv8.5. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.6 compliant if all of the following apply:

- It is Armv8.5 compliant.
- It includes all of the Armv8.6 architectural features that are mandatory.

An Armv8.6 compliant implementation can additionally include:

- Armv8.6 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.7, subject only to those constraints that require that certain features be implemented together.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AA32BF16, AArch32 BFloat16 instructions

FEAT_AA32BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch32 state. This format supports:

- Arithmetic instructions to multiply BFloat16 values and accumulate into single-precision results.
- Arithmetic instructions to accelerate dot product and matrix multiply-accumulate of BFloat16 values into single-precision results.
- Instructions to convert single-precision floating-point values to BFloat16 format.

This feature is supported in AArch32 state only.

FEAT_AA32BF16 is OPTIONAL from Armv8.2.

If FEAT_AA32BF16 is implemented, then FEAT_BF16 is implemented.

If FEAT_AA32BF16 is implemented, then FEAT_AA32 is implemented.

The following fields identify the presence of FEAT_AA32BF16:

- ID_ISAR6_EL1.BF16.
- ID_ISAR6.BF16.

For more information, see:

- BFloat16 floating-point format.
- Advanced SIMD BFloat16 instructions.
- Floating-point data-processing.

FEAT_AMUv1p1, Activity Monitors Extension version 1.1

FEAT_AMUv1p1 introduces support for virtualization of Activity Monitors event counters, and introduces controls to disable access to auxiliary event counters below the highest Exception level.

This feature is supported in AArch32 state and AArch64 state, if the hypervisor is using AArch64.

FEAT_AMUv1p1 is OPTIONAL from Armv8.5.

If FEAT_AMUv1p1 is implemented, then FEAT_AMUv1 is implemented.

The following fields identify the presence of FEAT_AMUv1p1:

- ID_AA64PFR0_EL1.AMU.
- ID_PFR0_EL1.AMU.
- ID_PFR0.AMU.
- EDPFR.AMU.

For more information, see The Activity Monitors Extension.

FEAT_BF16, AArch64 BFloat16 instructions

FEAT_BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch64 state. This format supports:

- Arithmetic instructions to multiply BFloat16 values and accumulate into single-precision results.
- Arithmetic instructions to accelerate dot product and matrix multiply-accumulate of BFloat16 values into single-precision results.
- Instructions to convert single-precision floating-point values to BFloat16 format.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in AArch64 state only.

FEAT_BF16 is OPTIONAL from Armv8.2.

In an Armv8.6 implementation, if FEAT_FP is implemented, FEAT_BF16 is implemented.

The following fields identify the presence of FEAT_BF16:

- ID_AA64ISAR1_EL1.BF16.
- ID_AA64ZFR0_EL1.BF16.

For more information, see:

- BFloat16 floating-point format.
- Summary of BFloat16 instruction behaviors.

FEAT_CP15SDISABLE2, CP15SDISABLE2

FEAT_CP15SDISABLE2 provides an implementation-defined mechanism, the CP15SDISABLE2 signal, which when asserted HIGH prevents writes to a set of Secure CP15 registers. This signal is analogous to the existing CP15SDISABLE signal.

This feature is supported only when EL3 is executing in AArch32 state.

FEAT_CP15SDISABLE2 is OPTIONAL from Armv8.0.

When FEAT_AA32 and FEAT_CP15SDISABLE2 are implemented, FEAT_AA32EL3 is implemented.

For more information, see The CP15SDISABLE and CP15SDISABLE2 input signals.

FEAT_DGH, Data Gathering Hint

FEAT_DGH introduces the Data Gathering Hint instruction to the hint space.

This instruction is added to the A64 instruction set only.

FEAT_DGH is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_DGH:

- ID_AA64ISAR1_EL1.DGH.

For more information, see Hint instructions.

FEAT_ECV, Enhanced Counter Virtualization

FEAT_ECV enhances the Generic Timer architecture. FEAT_ECV provides:

- Self-synchronizing views of the virtual and physical timers in AArch64 state and AArch32 state.
- The ability to scale the generation of the event stream when executing in AArch64 state or AArch32 state.
- When EL2 is using AArch64 state, traps to EL0 and EL1 access to the virtual counter or timer registers, and the physical timer registers when accessed using an EL02 mnemonic. The traps are configured in CNTHCTL_EL2, and apply to EL1 and EL0 accesses, whether EL1 and EL0 are in AArch64 state or AArch32 state.

For more information on the offset to views of physical time, see FEAT_ECV_POFF.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ECV is OPTIONAL from Armv8.5.

FEAT_ECV is mandatory from Armv8.6.

The following field identifies the presence of FEAT_ECV:

- ID_AA64MMFR0_EL1.ECV.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- Self-hosted trace timestamps.
- The profiling data.
- The AArch64 view of the Generic Timer.
- The AArch32 view of the Generic Timer.

FEAT_ECV_POFF, Enhanced Counter Virtualization Physical Offset

FEAT_ECV_POFF provides an offset between the EL1 or EL0 view of physical time, and the EL2 or EL3 view of physical time.

The offset to views of physical time at EL1 and EL0 apply in AArch64 state and AArch32 state.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ECV_POFF is OPTIONAL from Armv8.5.

If FEAT_ECV_POFF is implemented, then FEAT_ECV is implemented.

When FEAT_AA64 and FEAT_ECV_POFF are implemented, FEAT_AA64EL2 is implemented.

If FEAT_RME is implemented, then FEAT_ECV_POFF is implemented.

The following field identifies the presence of FEAT_ECV_POFF:

- ID_AA64MMFR0_EL1.ECV.

FEAT_FGT, Fine Grain Traps

FEAT_FGT introduces additional traps to EL2 of EL1 and EL0 access to individual or small groups of System registers and instructions, and traps to EL3 and EL2 of the Debug Communications Channel registers. The traps are independent of existing controls.

This feature is supported in AArch64, and when EL1 is using AArch64, EL0 accesses using AArch32 are also trapped.

FEAT_FGT is OPTIONAL from Armv8.5.

In an Armv8.6 implementation, if FEAT_AA64EL2 or FEAT_AA64EL3 is implemented, FEAT_FGT is implemented.

The following field identifies the presence of FEAT_FGT:

- ID_AA64MMFR0_EL1.FGT.

For more information, see Configurable instruction controls.

FEAT_HPMN0, Setting of MDCR_EL2.HPMN to zero

FEAT_HPMN0 permits a hypervisor to provide zero PMU event counters for a guest operating system by setting MDCR_EL2.HPMN to zero.

This feature is supported in both AArch64 and AArch32 states.

FEAT_HPMN0 is OPTIONAL from Armv8.5.

In an Armv8.8 implementation, if FEAT_PMUv3 and FEAT_EL2 are implemented, FEAT_HPMN0 is implemented.

If FEAT_HPMN0 is implemented, then FEAT_EL2 is implemented.

If FEAT_HPMN0 is implemented, then FEAT_PMUv3 and FEAT_FGT are implemented.

The following fields identify the presence of FEAT_HPMN0:

- ID_AA64DFR0_EL1.HPMN0.
- ID_DFR1_EL1.HPMN0.
- ID_DFR1.HPMN0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- Interaction with EL2.
- Controlling the PMU counters.
- The Performance Monitors Extension.
- The Performance Monitors Extension.

FEAT_MPAMv0p1, Memory System Resource Partitioning and Monitoring extension version 0.1

FEAT_MPAMv0p1 introduces support for version 0.1 of the MPAM extension.

This feature is supported in AArch64 state only.

FEAT_MPAMv0p1 is OPTIONAL from Armv8.5.

If FEAT_MPAMv0p1 is implemented, then FEAT_MPAM is implemented.

The following fields identify the presence of FEAT_MPAMv0p1:

- ID_AA64PFR0_EL1.MPAM.
- ID_AA64PFR1_EL1.MPAM_frac.

For more information, see MPAM PE Architecture.

FEAT_MPAMv1p1, Memory Partitioning and Monitoring extension version 1.1

FEAT_MPAMv1p1 introduces support for version 1.1 of the MPAM extension.

This feature is supported in AArch64 state only.

FEAT_MPAMv1p1 is OPTIONAL from Armv8.5.

If FEAT_MPAMv1p1 is implemented, then FEAT_MPAM is implemented.

If FEAT_MPAMv1p1 is implemented, then FEAT_MPAMv0p1 is not implemented.

The following fields identify the presence of FEAT_MPAMv1p1:

- ID_AA64PFR0_EL1.MPAM.
- ID_AA64PFR1_EL1.MPAM_frac.

For more information, see MPAM PE Architecture.

FEAT_MTPMU, Multi-threaded PMU extensions

FEAT_MTPMU introduces controls to disable PMEVTYPER<n>_EL0.MT.

From Armv8.6, when FEAT_PMUv3 is implemented, multithreaded event counting is only supported in multithreaded implementations that also include FEAT_MTPMU.

This feature is supported in both AArch64 and AArch32 states.

FEAT_MTPMU is OPTIONAL from Armv8.5.

If FEAT_MTPMU is implemented, then FEAT_PMUv3 is implemented.

If FEAT_MTPMU is implemented, then FEAT_EL2 or FEAT_EL3 is implemented.

The following fields identify the presence of FEAT_MTPMU:

- ID_AA64DFR0_EL1.MTPMU.
- ID_DFR1.MTPMU.
- ID_DFR1_EL1.MTPMU.

For more information, see:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Multithreaded implementations.
- MDCR_EL3.MTPME
- SDCR.MTPME
- MDCR_EL2.MTPME
- HDCR.MTPME.

FEAT_PAuth2, Enhancements to pointer authentication

FEAT_PAuth2 introduces enhanced pointer authentication functionality that changes the mechanism by which a PAC is added to the pointer.

This feature is supported in AArch64 state only.

FEAT_PAuth2 is OPTIONAL from Armv8.2.

FEAT_PAuth2 is mandatory from Armv8.6.

If FEAT_PAuth2 is implemented, then FEAT_PAuth is implemented.

If FEAT_PAuth2 is implemented, then FEAT_EPAC is not implemented.

The following fields identify the presence of FEAT_PAuth2:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

FEAT_TWED, Delayed Trapping of WFE

FEAT_TWED introduces support for configurable delayed trapping of the WFE and WFET instructions.

FEAT_TWED is OPTIONAL from Armv8.5.

The following field identifies the presence of FEAT_TWED:

- ID_AA64MMFR1_EL1.TWED.

For more information, see The Wait for Event and Wait for Event with Timeout instructions.

A2.2.7.1 Features added to the Armv8.6 extension in later releases

- FEAT_SPE_DPFZS.

A2.2.8 The Armv8.7 architecture extension

The Armv8.7 architecture extension is an extension to Armv8.6. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.7 compliant if all of the following apply:

- It is Armv8.6 compliant.
- It includes all of the Armv8.7 architectural features that are mandatory.

An Armv8.7 compliant implementation can additionally include:

- Armv8.7 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.8, subject only to those constraints that require that certain features be implemented together.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AFP, Alternate floating-point behavior

FEAT_AFP allows alternate behavior for specified floating-point instructions including:

- Flushing of denormalized numbers to zero can be controlled separately on inputs and outputs.
- Alternate NaN propagation rules and Default NaN values can apply.
- Certain scalar SIMD and floating-point instructions can be configured to preserve higher numbered SIMD vector elements.
- Changes to floating-point exception generation.

This feature is supported in AArch64 state only.

FEAT_AFP is OPTIONAL from Armv8.6.

In an Armv8.7 implementation, if FEAT_FP is implemented, FEAT_AFP is implemented.

If FEAT_AFP is implemented, then FEAT_FP is implemented.

The following field identifies the presence of FEAT_AFP:

- ID_AA64MMFR1_EL1.AFP.

For more information, see:

- Flushing denormalized numbers to zero.
- NaN handling and the Default NaN.
- Rounding.
- Floating-point exceptions and exception traps.

FEAT_EBF16, AArch64 Extended BFloat16 behaviors

FEAT_EBF16 supports the Extended BFloat16 behaviors.

This feature is supported in AArch64 state only.

FEAT_EBF16 is OPTIONAL from Armv8.2.

If FEAT_EBF16 is implemented, then FEAT_BF16 is implemented.

If FEAT_EBF16 is implemented, then FEAT_AdvSIMD, FEAT_FP, FEAT_SVE, or FEAT_SME is implemented.

The following fields identify the presence of FEAT_EBF16:

- ID_AA64ISAR1_EL1.BF16.
- ID_AA64ZFR0_EL1.BF16.

For more information, see:

- BFloat16 floating-point format.
- Floating-point behaviors for instructions that compute sum-of-products.

FEAT_HCX, Support for the HCRX_EL2 register

FEAT_HCX introduces the Extended Hypervisor Configuration Register, HCRX_EL2, that provides configuration controls for virtualization in addition to those provided by HCR_EL2, including defining whether various operations are trapped to EL2.

This feature is supported in AArch64 state only.

FEAT_HCX is OPTIONAL from Armv8.6.

In an Armv8.7 implementation, if FEAT_AA64EL2 is implemented, FEAT_HCX is implemented.

When FEAT_AA64 and FEAT_HCX are implemented, FEAT_AA64EL2 is implemented.

The following field identifies the presence of FEAT_HCX:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR1_EL1.HCX.

For more information, see Configurable instruction controls.

FEAT_LPA2, Larger physical address for 4KB and 16KB translation granules

FEAT_LPA2:

- Allows a larger VA space for each translation table base register of up to 52 bits when using the 4KB or 16KB translation granules.
- Allows a larger intermediate physical address (IPA) and PA space of up to 52 bits when using the 4KB or 16KB translation granules.
- Allows a level 0 block size where the block covers a 512GB address range for the 4KB translation granule if the implementation supports 52 bits of PA.
- Allows a level 1 block size where the block covers a 64GB address range for the 16KB translation granule if the implementation supports 52 bits of PA.

This feature is supported in AArch64 state only.

FEAT_LPA2 is OPTIONAL from Armv8.6.

If FEAT_LPA2 is implemented, then FEAT_LVA is implemented.

The following fields identify the presence of FEAT_LPA2:

- ID_AA64MMFR0_EL1.TGran4_2.
- ID_AA64MMFR0_EL1.TGran16_2.
- ID_AA64MMFR0_EL1.TGran4.
- ID_AA64MMFR0_EL1.TGran16.

For more information, see:

- Implemented physical address size.
- Output address size configuration.
- Supported virtual address ranges.
- Input address size configuration.
- Intermediate physical address size configuration.
- Translation using the 4KB granule.
- Translation using the 16KB granule.
- Table descriptor format.
- Block descriptor and Page descriptor formats.

FEAT_LS64, Support for 64-byte loads and stores without status

FEAT_LS64 introduces support for single-copy atomic 64-byte loads and stores without status result. For more information, see:

- LD64B.
- ST64B.

This feature is supported in AArch64 state only.

FEAT_LS64 is OPTIONAL from Armv8.6.

The following field identifies the presence of FEAT_LS64:

- ID_AA64ISAR1_EL1.LS64.

For more information, see Single-copy atomic 64-byte load/store.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LS64_ACCDATA, Support for 64-byte EL0 stores with status

FEAT_LS64_ACCDATA introduces support for single-copy atomic 64-byte EL0 stores with status result. For more information, see:

- ST64BV0.
- ACCDATA_EL1.

Note

The meaning of any status being returned by the ST64BV0 instruction is defined by the peripheral providing the response.

This feature is supported in AArch64 state only.

FEAT_LS64_ACCDATA is OPTIONAL from Armv8.6.

If FEAT_LS64_ACCDATA is implemented, then FEAT_LS64_V is implemented.

The following field identifies the presence of FEAT_LS64_ACCDATA:

- ID_AA64ISAR1_EL1.LS64.

For more information, see Single-copy atomic 64-byte load/store.

FEAT_LS64_V, Support for 64-byte stores with status

FEAT_LS64_V introduces support for single-copy atomic 64-byte stores with status result. For more information, see:

- ST64BV.

Note

The meaning of any status being returned by the ST64BV instruction is defined by the peripheral providing the response.

This feature is supported in AArch64 state only.

FEAT_LS64_V is OPTIONAL from Armv8.6.

If FEAT_LS64_V is implemented, then FEAT_LS64 is implemented.

The following field identifies the presence of FEAT_LS64_V:

- ID_AA64ISAR1_EL1.LS64.

For more information, see Single-copy atomic 64-byte load/store.

FEAT_MTE3, MTE Asymmetric Fault Handling

FEAT_MTE3 introduces support for asymmetric Tag Check Fault handling.

This feature is supported in AArch64 state only.

FEAT_MTE3 is OPTIONAL from Armv8.5.

In an Armv8.7 implementation, if FEAT_MTE_ASYNC is implemented, FEAT_MTE3 is implemented.

If FEAT_MTE3 is implemented, then FEAT_MTE2 is implemented.

The following field identifies the presence of FEAT_MTE3:

- ID_AA64PFR1_EL1.MTE.

For more information, see The Memory Tagging Extension.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_MTE_ASYM_FAULT, Memory tagging asymmetric faults

FEAT_MTE_ASYM_FAULT introduces support for asymmetric MTE Tag Check fault handling.

This feature is supported in AArch64 state only.

FEAT_MTE_ASYM_FAULT is OPTIONAL.

If FEAT_MTE3 is implemented, then FEAT_MTE_ASYM_FAULT is implemented.

If FEAT_MTE_ASYM_FAULT is implemented, then FEAT_MTE3 is implemented.

If FEAT_MTE_ASYM_FAULT is implemented, then FEAT_MTE_ASYNC is implemented.

The following field identifies the presence of FEAT_MTE_ASYM_FAULT:

- ID_AA64PFR1_EL1.MTE.

FEAT_PAN3, Support for SCTLR_ELx.EPAN

FEAT_PAN3 introduces a bit to SCTLR_EL1 and SCTLR_EL2, EPAN, to support using Privileged Access Never with instruction accesses for stage 1 translation regimes.

This feature is supported in AArch64 state only.

FEAT_PAN3 is OPTIONAL from Armv8.1.

FEAT_PAN3 is mandatory from Armv8.7.

If FEAT_PAN3 is implemented, then FEAT_PAN2 is implemented.

The following field identifies the presence of FEAT_PAN3:

- ID_AA64MMFR1_EL1.PAN.

For more information, see PSTATE PAN.

FEAT_PMUv3p7, Armv8.7 PMU extensions

FEAT_PMUv3p7 introduces the following to the Performance Monitors Extension:

- PMU counters can be frozen when an event counter has an unsigned overflow.
- Event counters can be prohibited from counting events at EL3 without affecting other Exception levels in Secure state.
- The cycle counter can be prohibited from counting cycles at EL3 without affecting other Exception levels in Secure state.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p7 is OPTIONAL from Armv8.6.

In an Armv8.7 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p7 is implemented.

If FEAT_PMUv3p7 is implemented, then FEAT_PMUv3p5 is implemented.

The following fields identify the presence of FEAT_PMUv3p7:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.
- PMCFGR.FZO.

For more information, see:

- Controlling the PMU counters.
- Freezing event counters.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_RPRES, Increased precision of FRECPE and FRSQRTE

FEAT_RPRES allows an increase in the precision of the single-precision floating-point reciprocal estimate and reciprocal square root estimate from an 8-bit mantissa to a 12-bit mantissa.

This feature is supported in AArch64 state only.

FEAT_RPRES is OPTIONAL from Armv8.6.

If FEAT_RPRES is implemented, then FEAT_AFP is implemented.

The following field identifies the presence of FEAT_RPRES:

- ID_AA64ISAR2_EL1.RPRES.

For more information, see `RecipEstimate()` and `RecipSqrtEstimate()`.

FEAT_SPE_FnE, Statistical Profiling inverse event filter

FEAT_SPE_FnE provides the capability to filter sample records by the inverse value of bits in the sampled Events packet.

This feature is supported in AArch64 state only.

FEAT_SPE_FnE is OPTIONAL.

If FEAT_SPE_FnE is implemented, then FEAT_SPEv1p2 is implemented.

If FEAT_SPEv1p2 is implemented, then FEAT_SPE_FnE is implemented.

The following field identifies the presence of FEAT_SPE_FnE:

- PMSIDR_EL1.FnE.

For more information, see Filtering sample records.

FEAT_SPE_PBT, Statistical Profiling previous branch target

FEAT_SPE_PBT provides support for generating a packet that provides the target address for the previous taken branch.

This feature is supported in AArch64 state only.

FEAT_SPE_PBT is OPTIONAL.

If FEAT_SPE_PBT is implemented, then FEAT_SPEv1p2 is implemented.

The following field identifies the presence of FEAT_SPE_PBT:

- PMSIDR_EL1.PBT.

For more information, see:

- Previous branch target.
- Address packet.

FEAT_SPEv1p2, Statistical Profiling Extensions version 1.2

FEAT_SPEv1p2 introduces the following to the Statistical Profiling Extension:

- Controls to freeze the PMU event counters after an SPE buffer management event occurs.
- A discard mode that allows all SPE data to be discarded rather than written to memory.

This feature is supported in AArch64 state only.

FEAT_SPEv1p2 is OPTIONAL from Armv8.6.

In an Armv8.7 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p2 is implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SPEv1p2 is implemented, then FEAT_SPEv1p1 is implemented.

The following field identifies the presence of FEAT_SPEv1p2:

- ID_AA64DFR0_EL1.PMSVer.

For more information, see:

- Freezing event counters.
- Discard mode.

FEAT_WFxT, WFE and WFI instructions with timeout

FEAT_WFxT introduces WFET and WFIT. These instructions support the generation of a local timeout event to act as a wake-up event for the PE when the virtual count in CNTVCT_EL0 equals or exceeds the value supplied by the instruction for the first time.

These instructions are added to the A64 instruction set only.

FEAT_WFxT is OPTIONAL from Armv8.6.

FEAT_WFxT is mandatory from Armv8.7.

The following field identifies the presence of FEAT_WFxT:

- ID_AA64ISAR2_EL1.WFxT.

For more information, see:

- Wait for Event.
- Wait for Interrupt mechanism.

FEAT_XS, XS attribute

FEAT_XS introduces the XS attribute for memory to indicate that an access could take a long time to complete. This feature provides variants of DSB instructions and TLB maintenance instructions, the completion of which does not depend on the completion of memory accesses with the XS attribute.

FEAT_XS adds:

- A mechanism to define the XS attribute for memory.
- An optional nXS variant to the AArch64 DSB instruction and OPTIONAL nXS qualifier to each AArch64 TLBI instruction to handle memory accesses with the XS attribute.
- The HCRX_EL2.FGTnXS bit to determine the behavior of fine-grained traps in HFGITR_EL2 for TLB maintenance instructions with the nXS qualifier.
- The HCRX_EL2.FnXS bit to determine the behavior of pre-existing TLB maintenance instructions in relation to the XS attribute.

This feature is supported in AArch64 state only, but the XS attribute also impacts AArch32 state execution.

FEAT_XS is OPTIONAL from Armv8.6.

FEAT_XS is mandatory from Armv8.7.

The following field identifies the presence of FEAT_XS:

- ID_AA64ISAR1_EL1.XS.

For more information, see:

- Data Synchronization Barrier (DSB).
- Behavior when stage 1 address translation is disabled.
- Block descriptor and Page descriptor formats.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Stage 1 memory type and Cacheability attributes.
- XS attribute modifier.
- Overview of memory region attributes for stage 1 translations.
- Ordering and completion of TLB maintenance instructions.

A2.2.8.1 Features added to the Armv8.7 extension in later releases

- FEAT_CSSC.
- FEAT_HAFT.
- FEAT_MTE4.
- FEAT_MTE_PERM.

A2.2.9 The Armv8.8 architecture extension

The Armv8.8 architecture extension is an extension to Armv8.7. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.8 compliant if all of the following apply:

- It is Armv8.7 compliant.
- It includes all of the Armv8.8 architectural features that are mandatory.

An Armv8.8 compliant implementation can additionally include:

- Armv8.8 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.9, subject only to those constraints that require that certain features be implemented together.

FEAT_CMOW, Control for cache maintenance permission

FEAT_CMOW introduces support for controlling the required permissions for some cache maintenance instructions that operate by VA and perform translation such that:

- Stage 1 can be configured to generate a Permission fault if write permission is not present for cache maintenance instructions executed at EL0.
- Stage 2 can be configured to generate a Permission fault if write permission is not present for cache maintenance instructions executed at EL1 or EL0.

This feature is supported in AArch64 state only, but also impacts AArch32 instructions.

FEAT_CMOW is OPTIONAL from Armv8.7.

FEAT_CMOW is mandatory from Armv8.8.

The following field identifies the presence of FEAT_CMOW:

- ID_AA64MMFR1_EL1.CMOW.

For more information, see:

- A64 Cache maintenance instructions.
- Permission fault.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_Debugv8p8, Debug v8.8

FEAT_Debugv8p8 introduces support to allow an asynchronous exception to be taken after an exception generates an Exception Catch debug event, but before the PE halts.

This feature is supported in both AArch64 and AArch32 states.

FEAT_Debugv8p8 is OPTIONAL from Armv8.7.

FEAT_Debugv8p8 is mandatory from Armv8.8.

If FEAT_Debugv8p8 is implemented, then FEAT_Debugv8p4 is implemented.

The following fields identify the presence of FEAT_Debugv8p8:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- DBGDIDR.Version.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

For more information, see Exception Catch debug event.

FEAT_HBC, Hinted conditional branches

FEAT_HBC provides the BC.cond instruction to give a conditional branch to a label at a PC-relative offset, with a hint that this branch is very unlikely to change direction.

This feature is supported in AArch64 state only.

FEAT_HBC is OPTIONAL from Armv8.7.

FEAT_HBC is mandatory from Armv8.8.

The following field identifies the presence of FEAT_HBC:

- ID_AA64ISAR2_EL1.BC.

For more information, see Conditional branch.

FEAT_MOPS, Standardization of memory operations

FEAT_MOPS provides instructions that perform a memory copy or memory set, and introduces Memory Copy and Memory Set exceptions.

FEAT_MOPS also introduces the HCRX_EL2.{MSCEn, MCE2}, SCTLR_EL1.MSCEn, and SCTLR_EL2.MSCEn control bits.

This feature is supported in AArch64 state only.

FEAT_MOPS is OPTIONAL from Armv8.7.

FEAT_MOPS is mandatory from Armv8.8.

The following field identifies the presence of FEAT_MOPS:

- ID_AA64ISAR2_EL1.MOPS.

For more information, see:

- Memory Copy and Memory Set instructions.
- Memory Copy and Memory Set exceptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_NMI, Non-maskable Interrupts

FEAT_NMI provides a mechanism to support non-maskable interrupts (NMI) and less-masked interrupts (LMI). In addition to legacy behavior, the feature includes the following:

- A mode for supporting an LMI interrupt mask that is distinct from PSTATE.{I, F}.
- A mode for supporting a limited NMI, where the value when PSTATE.SP is 1 is taken as an interrupt mask for all interrupts targeting that Exception level, and where the LMI interrupt mask can also be used

FEAT_NMI adds:

- The AllIntMask variable.
- An Optional Superpriority attribute to denote virtual and physical IRQ and FIQ interrupts as non-maskable.
- The SCTLR_ELx.{NMI, SPINTMASK} control bits.
- The PSTATE.ALLINT bit and associated instructions.
- The HCRX_EL2.TALLINT bit to enable trapping of ALLINT instructions at EL1.

This feature is supported in AArch64 state only.

FEAT_NMI is OPTIONAL from Armv8.7.

FEAT_NMI is mandatory from Armv8.8.

The following field identifies the presence of FEAT_NMI:

- ID_AA64PFR1_EL1.NMI.

For more information, see:

- Asynchronous exception types.
- Virtual interrupts.
- PSTATE fields that are meaningful in AArch64 state.
- WFE wake-up events.

FEAT_PMUv3_EXT64, 64-bit external interface to the Performance Monitors

FEAT_PMUv3_EXT64 indicates the external Performance Monitors registers are implemented as 64-bit registers. The 32-bit CoreSight management registers remain 32-bit registers.

FEAT_PMUv3_EXT64 is OPTIONAL from Armv8.8.

If FEAT_PMUv3_EXT64 is implemented, then FEAT_PMUv3_EXT is implemented.

If FEAT_PMUv3_EXT64 is implemented, then FEAT_PMUv3_EXT32 is not implemented.

The following field identifies the presence of FEAT_PMUv3_EXT64:

- PMDEVARCH.ARCHPART.

For more information, see Recommended External Interface to the Performance Monitors.

FEAT_PMUv3_TH, Event counting threshold

FEAT_PMUv3_TH introduces threshold condition controls to each PMEVTYPER<n>_EL0 register. This feature permits the counter to count only when PMEVTYPER<n>.{MT, evtCount} describe an event whose count meets a specified threshold condition.

This feature is supported in both AArch64 and AArch32 states. The threshold condition controls are accessible only in AArch64 state. However, threshold conditions still apply in AArch32 state.

FEAT_PMUv3_TH is OPTIONAL from Armv8.7.

If FEAT_PMUv3_TH is implemented, then FEAT_PMUv3 is implemented.

The following fields identify the presence of FEAT_PMUv3_TH:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMMIR_EL1.THWIDTH.
- PMMIR.THWIDTH.
- PMMIR.THWIDTH.

For more information, see Event threshold and edge counting.

FEAT_PMUv3p8, Armv8.8 PMU extensions

FEAT_PMUv3p8 introduces the following to the Performance Monitors Extension:

- The Common event number space is extended to include the ranges `0x0040-0x00BF` and `0x4040-0x40BF`.
- For an event counter n, if any reserved or unimplemented PMU event number is written to PMEVTYPER<n>.evtCount, then event counter n does not count, and a read of PMEVTYPER<n>.evtCount returns the value written.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p8 is OPTIONAL from Armv8.7.

In an Armv8.8 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p8 is implemented.

If FEAT_PMUv3p8 is implemented, then FEAT_PMUv3p7 is implemented.

The following fields identify the presence of FEAT_PMUv3p8:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see PMU Event Descriptions.

FEAT_SCTLR2, Extension to SCTLR_ELx

FEAT_SCTLR2 introduces the SCTLR2_ELx registers, which provide top-level control of the system, including its memory system. These registers are extensions of the corresponding SCTLR_ELx registers.

This feature is supported in AArch64 state only.

FEAT_SCTLR2 is OPTIONAL from Armv8.0.

FEAT_SCTLR2 is mandatory from Armv8.9.

When FEAT_SCTLR2 and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following field identifies the presence of FEAT_SCTLR2:

- ID_AA64MMFR3_EL1.SCTLRX.

FEAT_SPEv1p3, Statistical Profiling Extensions version 1.3

FEAT_SPEv1p3 introduces the following to the Statistical Profiling Extension:

- Support for sampling Tag operations.
- Support for sampling Memory Copy and Set operations.

This feature is supported in AArch64 state only.

FEAT_SPEv1p3 is OPTIONAL from Armv8.7.

In an Armv8.8 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p3 is implemented.

If FEAT_SPEv1p3 is implemented, then FEAT_SPEv1p2 is implemented.

The following field identifies the presence of FEAT_SPEv1p3:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64DFR0_EL1.PMSVer.

For more information, see:

- Additional information for each profiled memory access operation.
- About the Statistical Profiling Extension sample records.
- Address packet.

FEAT_TCR2, Support for TCR2_ELx

FEAT_TCR2 introduces the TCR2_ELx registers which provide top-level control of the EL1&0 and EL2&0 translation regimes respectively. These registers are extensions of the corresponding TCR_ELx registers.

This feature is supported in AArch64 state only.

FEAT_TCR2 is OPTIONAL from Armv8.0.

FEAT_TCR2 is mandatory from Armv8.9.

When FEAT_TCR2 and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following field identifies the presence of FEAT_TCR2:

- ID_AA64MMFR3_EL1.TCRX.

FEAT_TIDCP1, EL0 use of IMPLEMENTATION DEFINED functionality

FEAT_TIDCP1 introduces a control at EL1 and EL2 to enable trapping of EL0 accesses to registers that might control IMPLEMENTATION DEFINED functions.

This feature introduces controls only in AArch64 state, and controls IMPLEMENTATION DEFINED execution at EL0 in both AArch32 and AArch64 states.

FEAT_TIDCP1 is OPTIONAL from Armv8.7.

FEAT_TIDCP1 is mandatory from Armv8.8.

The following field identifies the presence of FEAT_TIDCP1:

- ID_AA64MMFR1_EL1.TIDCP1.

For more information, see Prioritization of Synchronous exceptions taken to AArch64 state

A2.2.10 The Armv8.9 architecture extension

The Armv8.9 architecture extension is an extension to Armv8.8. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.9 compliant if all of the following apply:

- It is Armv8.8 compliant.
- It includes all of the Armv8.9 architectural features that are mandatory.

An Armv8.9 compliant implementation can additionally include:

- Armv8.9 features that are optional.

FEAT_ADERR, Asynchronous Device Error Exceptions

FEAT_ADERR introduces controls for whether an error signaled on a load from Device memory is handled precisely and synchronously.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ADERR is OPTIONAL from Armv8.8.

If FEAT_ADERR is implemented, then FEAT_RASv2 is implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_ADERR is implemented, then FEAT_SCTLR2 is implemented.

When FEAT_AA64 and FEAT_ADERR are implemented, FEAT_AA64EL1 is implemented.

When FEAT_ADERR and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following fields identify the presence of FEAT_ADERR:

- ID_AA64MMFR3_EL1.ADERR.
- ID_AA64MMFR3_EL1.SDERR.

For more information, see:

- About the RAS Extension.
- Taking error exceptions.

FEAT_AIE, Memory Attribute Index Enhancement

FEAT_AIE increases the stage 1 descriptor attribute index bit width from 3 to 4, allowing use of up to 16 memory attributes.

This feature is supported in AArch64 state only.

FEAT_AIE is OPTIONAL from Armv8.8.

If FEAT_AIE is implemented, then FEAT_TCR2 is implemented.

If FEAT_AIE is implemented, then FEAT_HPDS is implemented.

The following field identifies the presence of FEAT_AIE:

- ID_AA64MMFR3_EL1.AIE.

For more information, see Stage 1 memory type and Cacheability attributes.

FEAT_AMU_EXT64, the 64-bit external Activity Monitors extension

FEAT_AMU_EXT64 indicates the external AMU registers are implemented as 64-bit registers.

FEAT_AMU_EXT64 is OPTIONAL.

If FEAT_AMU_EXT64 is implemented, then FEAT_AMU_EXT is implemented.

If FEAT_AMU_EXT64 is implemented, then FEAT_AMU_EXT32 is not implemented.

The following field identifies the presence of FEAT_AMU_EXT64:

- AMDEVARCH.ARCHID.

For more information, see Recommended External Interface to the Activity Monitors.

FEAT_ANERR, Asynchronous Normal Error Exceptions

FEAT_ANERR introduces controls for whether an error signaled on a load from Normal memory is handled precisely and synchronously.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ANERR is OPTIONAL from Armv8.8.

If FEAT_ANERR is implemented, then FEAT_RASv2 is implemented.

If FEAT_ANERR is implemented, then FEAT_SCTLR2 is implemented.

When FEAT_AA64 and FEAT_ANERR are implemented, FEAT_AA64EL1 is implemented.

When FEAT_ANERR and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following fields identify the presence of FEAT_ANERR:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR3_EL1.ANERR.
- ID_AA64MMFR3_EL1.SNERR.

For more information, see:

- About the RAS Extension.
- Taking error exceptions.

FEAT_ATS1A, Address Translation operations that ignore stage 1 permissions

FEAT_ATS1A introduces instructions that provide the output address and attributes of a valid translation without checking for stage 1 permissions.

These instructions are added to the A64 instruction set only.

FEAT_ATS1A is OPTIONAL from Armv8.8.

The following field identifies the presence of FEAT_ATS1A:

- ID_AA64ISAR2_EL1.ATS1A.

FEAT_CLRBHB, Support for Clear Branch History instruction

FEAT_CLRBHB provides a CLRBHB instruction, which can be used to clear the branch history.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CLRBHB is OPTIONAL from Armv8.0.

FEAT_CLRBHB is mandatory from Armv8.9.

The following fields identify the presence of FEAT_CLRBHB:

- ID_AA64ISAR2_EL1.CLRBHB.
- ID_ISAR6_EL1.CLRBHB.
- ID_ISAR6.CLRBHB.

For more information, see:

- Branch prediction.
- AArch32 cache and branch predictor maintenance instructions.

FEAT_CSSC, Common Short Sequence Compression instructions

FEAT_CSSC introduces a set of instructions for optimization of short instruction sequences using general-purpose registers.

This feature is supported in AArch64 state only.

FEAT_CSSC is OPTIONAL from Armv8.7.

FEAT_CSSC is mandatory from Armv8.9.

The following field identifies the presence of FEAT_CSSC:

- ID_AA64ISAR2_EL1.CSSC.

For more information, see:

- Integer minimum and maximum (immediate).
- Integer maximum and minimum (register).
- Absolute value.
- Bit operation.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_Debugv8p9, Debug v8.9

FEAT_Debugv8p9 introduces all of the following:

- The ability to implement more than 16 breakpoints.
- The ability to implement more than 16 watchpoints.
- DBGBCR<n>_EL1 and DBGWCR<n>_EL1 are 64-bit registers in the external debug interface.
- DSPSR2 is added to extend DSPSR for holding the saved process state for Debug state.

This feature is supported in both AArch64 and AArch32 states.

FEAT_Debugv8p9 is OPTIONAL from Armv8.8.

FEAT_Debugv8p9 is mandatory from Armv8.9.

If FEAT_Debugv8p9 is implemented, then FEAT_Debugv8p8 is implemented.

When FEAT_Debugv8p9 and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_Debugv8p9:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- ID_DFR0.CopDbg.
- DBGDIDR.Version.
- EDDEVARCH.ARCHVER.
- EDDEVID1.HSR.

For more information, see:

- About Breakpoint exceptions.
- About Watchpoint exceptions.

FEAT_DoubleFault2, Double Fault Extension v2

FEAT_DoubleFault2 provides additional controls for routing and masking error exceptions.

This feature is supported in AArch64 state only.

FEAT_DoubleFault2 is OPTIONAL from Armv8.8.

If FEAT_DoubleFault2 is implemented, then FEAT_SCTLR2 is implemented.

When FEAT_AA64 and FEAT_DoubleFault2 are implemented, FEAT_AA64EL1 is implemented.

When FEAT_DoubleFault2 and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

If FEAT_DoubleFault2 is implemented, then FEAT_DoubleFault is implemented.

The following field identifies the presence of FEAT_DoubleFault2:

- ID_AA64PFR1_EL1.DF2.

For more information, see:

- Synchronous exception types.
- Asynchronous exception types.
- Error synchronization event.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ECBHB, Exploitative control using branch history information

FEAT_ECBHB imposes restrictions on branch history speculation around exceptions.

This feature is supported in AArch64 state only.

FEAT_ECBHB is OPTIONAL from Armv8.0.

FEAT_ECBHB is mandatory from Armv8.9.

The following field identifies the presence of FEAT_ECBHB:

- ID_AA64MMFR1_EL1.ECBHB.

For more information, see Branch prediction.

FEAT_EDHSR, Support for EDHSR

FEAT_EDHSR introduces the EDHSR, which holds syndrome information of a Debug event.

FEAT_EDHSR is OPTIONAL.

If FEAT_Debugv8p9 is implemented, then FEAT_EDHSR is implemented.

If FEAT_EDHSR is implemented, then FEAT_Debugv8p2 is implemented.

The following field identifies the presence of FEAT_EDHSR:

- EDDEVID1.HSR.

FEAT_FGT2, Fine-grained traps 2

FEAT_FGT2 introduces the hypervisor registers HFGITR2_EL2, HFGRTR2_EL2, HFGWTR_EL2, HDFGRTR2_EL2, and HDFGWTR2_EL2. These registers are extensions of the corresponding FGT registers.

This feature is supported in AArch64, and in AArch32 at EL0 when EL1 is using AArch64.

FEAT_FGT2 is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_AA64EL2 is implemented, FEAT_FGT2 is implemented.

If FEAT_FGT2 is implemented, then FEAT_FGT is implemented.

The following field identifies the presence of FEAT_FGT2:

- ID_AA64MMFR0_EL1.FGT.

FEAT_HAFT, Hardware managed Access Flag for Table descriptors

FEAT_HAFT introduces the support for hardware management of the Table descriptor Access flag.

This feature is supported in AArch64 state only.

FEAT_HAFT is OPTIONAL from Armv8.7.

If FEAT_HAFT is implemented, then FEAT_HAFDBS is implemented.

If FEAT_HAFT is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_HAFT:

- ID_AA64MMFR1_EL1.HAFDBS.

For more information, see Hardware management of the Table descriptor Access Flag.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LRCPC3, Load-Acquire RCpc instructions version 3

FEAT_LRCPC3 introduces variants of load/store pair and load/store single register instructions, with release consistency, to optimize additional use cases where ordering is required. FEAT_LRCPC3 also introduces a set of additional load/store instructions with release consistency ordering in the Advanced SIMD and floating-point instruction set.

This feature is supported in AArch64 state only.

FEAT_LRCPC3 is OPTIONAL from Armv8.2.

If FEAT_LRCPC3 is implemented, then FEAT_LRCPC2 is implemented.

The following field identifies the presence of FEAT_LRCPC3:

- ID_AA64ISAR1_EL1.LRCPC.

For more information, see:

- Changes to single-copy atomicity in Armv8.4.
- Load-Acquire/Store-Release.
- A64 instructions that are changed in Debug state.

FEAT_MTE4, Enhanced Memory Tagging Extension

FEAT_MTE4 introduces support for the following sub-features:

- Canonical tag checking, identified as FEAT_MTE_CANONICAL_TAGS.
- Reporting of all non-address bits on a fault, identified as FEAT_MTE_TAGGED_FAR.
- Store-only Tag checking, identified as FEAT_MTE_STORE_ONLY.
- Memory tagging with Address tagging disabled, identified as FEAT_MTE_NO_ADDRESS_TAGS.

This feature is supported in AArch64 state only.

FEAT_MTE4 is OPTIONAL from Armv8.7.

In an Armv8.9 implementation, if FEAT_MTE2 is implemented, FEAT_MTE4 is implemented.

If FEAT_MTE4 is implemented, then FEAT_MTE2 is implemented.

If FEAT_MTE4 is implemented, then FEAT_MTE_PERM is implemented.

If FEAT_MTE4 is implemented, then FEAT_MTE_CANONICAL_TAGS, FEAT_MTE_NO_ADDRESS_TAGS, FEAT_MTE_TAGGED_FAR, and FEAT_MTE_STORE_ONLY are implemented.

FEAT_MTE_ASYNC, Asynchronous reporting of Tag Check Fault

FEAT_MTE_ASYNC provides support for asynchronously accumulating Tag Check Faults into the TFSRE0_EL1 or TFSR_ELx registers. A PE that is compliant with FEAT_MTE2 is compliant with the behavior defined for this feature.

This feature is supported in AArch64 state only.

FEAT_MTE_ASYNC is OPTIONAL from Armv8.5.

If FEAT_MTE_ASYNC is implemented, then FEAT_MTE2 is implemented.

The following field identifies the presence of FEAT_MTE_ASYNC:

- ID_AA64PFR1_EL1.MTE_frac.

For more information, see:

- The Memory Tagging Extension.
- The AArch64 Application Level Memory Model.
- PMU Event Descriptions.
- The Statistical Profiling Extension.
- Debug State.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_MTE_CANONICAL_TAGS, Canonical Tag checking for Untagged memory

FEAT_MTE_CANONICAL_TAGS introduces support for MTE canonical tag checking.

This feature is supported in AArch64 state only.

FEAT_MTE_CANONICAL_TAGS is OPTIONAL.

If FEAT_MTE_CANONICAL_TAGS is implemented, then FEAT_MTE4 is implemented.

The following field identifies the presence of FEAT_MTE_CANONICAL_TAGS:

- ID_AA64PFR1_EL1.MTEX.

FEAT_MTE_NO_ADDRESS_TAGS, Memory tagging with Address tagging disabled

FEAT_MTE_NO_ADDRESS_TAG introduces support for MTE tagging with Address tagging disabled.

This feature is supported in AArch64 state only.

FEAT_MTE_NO_ADDRESS_TAGS is OPTIONAL.

If FEAT_MTE_NO_ADDRESS_TAGS is implemented, then FEAT_MTE4 is implemented.

The following field identifies the presence of FEAT_MTE_NO_ADDRESS_TAGS:

- ID_AA64PFR1_EL1.MTEX.

FEAT_MTE_PERM, Allocation tag access permission

FEAT_MTE_PERM introduces support for the Stage 2 NoTagAccess memory attribute.

This feature is supported in AArch64 state only.

FEAT_MTE_PERM is OPTIONAL from Armv8.7.

In an Armv8.9 implementation, if FEAT_MTE2 is implemented, FEAT_MTE_PERM is implemented.

If FEAT_MTE_PERM is implemented, then FEAT_MTE2 is implemented.

The following field identifies the presence of FEAT_MTE_PERM:

- ID_AA64PFR2_EL1.MTEPERM.

FEAT_MTE_STORE_ONLY, Store-only Tag Checking

This feature is supported in AArch64 state only.

FEAT_MTE_STORE_ONLY is OPTIONAL.

If FEAT_MTE_STORE_ONLY is implemented, then FEAT_MTE4 is implemented.

The following field identifies the presence of FEAT_MTE_STORE_ONLY:

- ID_AA64PFR2_EL1.MTESTOREONLY.

FEAT_MTE_TAGGED_FAR, FAR_ELx on a Tag Check Fault

FEAT_MTE_TAGGED_FAR introduces support for reporting all non-address bits on a synchronous MTE tag check fault.

This feature is supported in AArch64 state only.

FEAT_MTE_TAGGED_FAR is OPTIONAL.

If FEAT_MTE_TAGGED_FAR is implemented, then FEAT_MTE4 is implemented.

The following field identifies the presence of FEAT_MTE_TAGGED_FAR:

- ID_AA64PFR2_EL1.MTEFAR.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PCSRv8p9, Armv8.9 PC Sample-based Profiling Extension

FEAT_PCSRv8p9 introduces a mechanism to suspend PC Sample-based Profiling.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PCSRv8p9 is OPTIONAL from Armv8.8.

If FEAT_PCSRv8p9 is implemented, then FEAT_PCSRv8p2 is implemented.

The following field identifies the presence of FEAT_PCSRv8p9:

- PMDEVID.PCSample.

For more information, see Suspending and activating PC Sample-based Profiling.

FEAT_PFAR, Physical Fault Address Register Extension

FEAT_PFAR introduces the Physical Fault Address Registers, PFAR_ELx, that record the faulting physical address for a synchronous External abort or SError exception.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PFAR is OPTIONAL from Armv8.8.

When FEAT_AA64 and FEAT_PFAR are implemented, FEAT_AA64EL1 is implemented.

When FEAT_PFAR and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_PFAR:

- ID_AA64PFR1_EL1.PFAR.

FEAT_PMUv3_EDGE, PMU event edge detection

FEAT_PMUv3_EDGE introduces edge-detection logic to support counting threshold crossing events.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3_EDGE is OPTIONAL from Armv8.8.

If FEAT_PMUv3_EDGE is implemented, then FEAT_PMUv3_TH is implemented.

The following fields identify the presence of FEAT_PMUv3_EDGE:

- PMMIR_EL1.EDGE.
- PMMIR.EDGE.
- PMMIR.EDGE.

For more information, see Event threshold and edge counting.

FEAT_PMUv3_ICNTR, Fixed-function instruction counter

FEAT_PMUv3_ICNTR introduces a fixed-function instruction counter to the PMU.

This feature is supported in both AArch64 and AArch32 states. The counter is not accessible from AArch32 state.

FEAT_PMUv3_ICNTR is OPTIONAL from Armv8.8.

If FEAT_PMUv3_ICNTR is implemented, then FEAT_PMUv3p9 is implemented.

When FEAT_PMUv3_ICNTR and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_PMUv3_ICNTR:

- ID_AA64DFR1_EL1.PMICNTR.
- PMCFGR.NCG.

For more information, see About the Performance Monitors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PMUv3_SS, PMU Snapshot extension

FEAT_PMUv3_SS defines an IMPLEMENTATION DEFINED Snapshot Extension, compatible with the CoreSight PMU Snapshot Extension.

This feature is supported in both AArch64 and AArch32 states. The PMU snapshot registers are not accessible from AArch32 state.

FEAT_PMUv3_SS is OPTIONAL from Armv8.8.

If FEAT_PMUv3_SS is implemented, then FEAT_PMUv3p9 is implemented.

When FEAT_PMUv3_SS and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

If FEAT_PMUv3_SS is implemented, then FEAT_AA32EL1 is not implemented.

The following fields identify the presence of FEAT_PMUv3_SS:

- ID_AA64DFR0_EL1.PMSS.
- PMDEVID.PMSS.

For more information, see PMU snapshots.

FEAT_PMUv3p9, Armv8.9 PMU extensions

FEAT_PMUv3p9 introduces the following to the Performance Monitors Extension:

- Provides finer-grained control over allocation of PMU event counters to an EL0 process.
- Allows an arbitrary combination of event counters and fixed-function counters to be zeroed.
- Provides controls to configure the PMU to directly request the PE enters Debug state without using the CTI.
- Updates PMU event definitions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p9 is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p9 is implemented.

If FEAT_PMUv3p9 is implemented, then FEAT_PMUv3p8 is implemented.

When FEAT_PMUv3p9 and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_PMUv3p9:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:

- PMU Overflow external debug request.
- EL0 access controls.
- PMU Event Descriptions.
- Resetting counters.

FEAT_PRFMSLC, SLC target support for PRFM and PRFUM instructions

FEAT_PRFMSLC introduces a system level cache option for the PRFM and PRFUM instructions.

This feature is supported in AArch64 state only.

FEAT_PRFMSLC is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_PRFMSLC:

- ID_AA64ISAR2_EL1.PRFMSLC.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_RASv2, RAS Extension v2

FEAT_RASv2 introduces the following to the Reliability, Availability, and Serviceability Extension:

- The features defined by FEAT_RASSAv2 in the System register error records.
- The error group status register, ERXGSR_EL1.
- A control to trap writes to RAS error record System registers to EL3.
- An additional syndrome to ESR_ELx for error exceptions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_RASv2 is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_RAS is implemented, FEAT_RASv2 is implemented.

When FEAT_RASv2 and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

If FEAT_RASv2 is implemented, then FEAT_RASv1p1 is implemented.

If FEAT_RASv2 is implemented, then FEAT_RASSAv2 is implemented.

The following fields identify the presence of FEAT_RASv2:

- ID_AA64PFR0_EL1.RAS.
- ID_PFR0_EL1.RAS.
- ID_PFR0.RAS.

For more information, see:

- RAS PE architecture.
- *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

FEAT_RPRFM, Support for Range Prefetch Memory instruction

FEAT_RPRFM introduces the Range Prefetch Memory hint instruction, RPRFM which specifies the range of addresses that are likely to be accessed in the near future and their expected reuse.

This feature is supported in AArch64 state only.

FEAT_RPRFM is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_RPRFM:

- ID_AA64ISAR2_EL1.RPRFM.

FEAT_S1PIE, Stage 1 permission indirections

FEAT_S1PIE introduces a way to set stage 1 permissions that allows more efficient use of the permission bits in translation table descriptors and provides the ability to introduce additional permission types.

This feature is supported in AArch64 state only.

FEAT_S1PIE is OPTIONAL from Armv8.8.

If FEAT_S1PIE is implemented, then FEAT_ATS1A is implemented.

If FEAT_S1PIE is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_S1PIE:

- ID_AA64MMFR3_EL1.S1PIE.

For more information, see Stage 1 Indirect permissions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_S1POE, Stage 1 permission overlays

FEAT_S1POE allows stage 1 permissions to be progressively restricted by processes running at EL0 without requiring TLB maintenance, and reduces the number of calls to privileged software.

This feature is supported in AArch64 state only.

FEAT_S1POE is OPTIONAL from Armv8.8.

If FEAT_S1POE is implemented, then FEAT_TCR2 is implemented.

If FEAT_S1POE is implemented, then FEAT_ATS1A is implemented.

If FEAT_S1POE is implemented, then FEAT_HPDS is implemented.

The following field identifies the presence of FEAT_S1POE:

- ID_AA64MMFR3_EL1.S1POE.

For more information, see Stage 1 Overlay permissions.

FEAT_S2PIE, Stage 2 permission indirections

FEAT_S2PIE introduces all of the following:

- A method to set stage 2 permissions that allows more efficient use of the permission bits in translation table descriptors and provides the ability to introduce additional permission types.
- The Mostly Read Only (MRO) permission in stage 2 translations.

This feature is supported in AArch64 state only.

FEAT_S2PIE is OPTIONAL from Armv8.8.

If FEAT_S2PIE is implemented, then FEAT_EL2 is implemented.

The following field identifies the presence of FEAT_S2PIE:

- ID_AA64MMFR3_EL1.S2PIE.

For more information, see Stage 2 Indirect permissions.

FEAT_S2POE, Stage 2 permission overlays

FEAT_S2POE provides a mechanism for stage 2 permissions to be progressively restricted, such that different EL1&0 contexts that are using the same stage 2 translation tables can be provided with different sets of permissions.

This feature is supported in AArch64 state only.

FEAT_S2POE is OPTIONAL from Armv8.8.

If FEAT_S2POE is implemented, then FEAT_EL2 is implemented.

If FEAT_S2POE is implemented, then FEAT_S2PIE is implemented.

The following field identifies the presence of FEAT_S2POE:

- ID_AA64MMFR3_EL1.S2POE.

For more information, see Stage 2 Overlay permissions.

FEAT_SPECRES2, Enhanced speculation restriction instructions

FEAT_SPECRES2 introduces a speculation restriction instruction, Clear Other Speculative Prediction Restriction by Context (COSP), to the instructions that are part of FEAT_SPECRES.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SPECRES2 is OPTIONAL from Armv8.0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPECRES2 is mandatory from Armv8.9.

The following fields identify the presence of FEAT_SPECRES2:

- ID_AA64ISAR1_EL1.SPECRES.
- ID_ISAR6_EL1.SPECRES.
- ID_ISAR6.SPECRES.

For more information, see:

- Prediction restriction instructions.
- Execution, data prediction and prefetching restriction System instructions.
- Execution and data prediction restriction System instructions.

FEAT_SPE_CRR, Statistical Profiling call return branch records

FEAT_SPE_CRR extends the Operation Type packet to provide more information whether the branch is a procedure call or a procedure return.

This feature is supported in AArch64 state only.

FEAT_SPE_CRR is OPTIONAL.

In an Armv8.9 implementation, if FEAT_SPE is implemented, FEAT_SPE_CRR is implemented.

If FEAT_SPE_CRR is implemented, then FEAT_SPEv1p4 is implemented.

If FEAT_SPE and FEAT_GCS are implemented, then FEAT_SPE_CRR is implemented.

The following field identifies the presence of FEAT_SPE_CRR:

- PMSIDR_EL1.CRR.

FEAT_SPE_DPFZS, Disable Cycle Counter on SPE Freeze

FEAT_SPE_DPFZS introduces controls to disable cycle counting when event counting is frozen on a Statistical Profiling Buffer Management event.

This feature is supported in AArch64 state only.

FEAT_SPE_DPFZS is OPTIONAL from Armv8.6.

If FEAT_PMUv3p9 and FEAT_SPEv1p4 are implemented, then FEAT_SPE_DPFZS is implemented.

If FEAT_SPE_DPFZS is implemented, then FEAT_PMUv3p7 and FEAT_SPEv1p2 are implemented.

The following field identifies the presence of FEAT_SPE_DPFZS:

- ID_AA64DFR1_EL1.DPFZS.

FEAT_SPE_FDS, Statistical Profiling data source filtering

FEAT_SPE_FDS provides the capability to filter sample records by all or part of the sampled Data Source packet.

This feature is supported in AArch64 state only.

FEAT_SPE_FDS is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_SPE_LDS is implemented, FEAT_SPE_FDS is implemented.

If FEAT_SPE_FDS is implemented, then FEAT_SPE_LDS is implemented.

If FEAT_SPE_FDS is implemented, then FEAT_SPEv1p4 is implemented.

When FEAT_SPE_FDS and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_SPE_FDS:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMSIDR_EL1.FDS.

For more information, see Filtering sample records.

FEAT_SPEv1p4, Statistical Profiling Extension version 1.4

FEAT_SPEv1p4 introduces the following to the Statistical Profiling Extension:

- Additions to the Events packet to provide more information about the data source.
- Additional event-based record filtering control bits in PMSEVFR_EL1 and PMSNEVFR_EL1 that add the ability to filter by additional existing bits in the Events packet.

This feature is supported in AArch64 state only.

FEAT_SPEv1p4 is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p4 is implemented.

If FEAT_SPEv1p4 is implemented, then FEAT_SPEv1p3 is implemented.

The following field identifies the presence of FEAT_SPEv1p4:

- ID_AA64DFR0_EL1.PMSVer.

FEAT_SPMU, System Performance Monitors Extension

FEAT_SPMU provides a framework of architectural System registers and behaviors for System PMUs, which are PMUs other than the Performance Monitors Extension PMU, that are accessible by the PE.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SPMU is OPTIONAL from Armv8.8.

If FEAT_SPMU is implemented, then FEAT_PMUv3p9 is implemented.

When FEAT_SPMU and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_SPMU:

- ID_AA64DFR1_EL1.SPMU.

For more information, see System Performance Monitors Extension.

FEAT_THE, Translation Hardening Extension

FEAT_THE provides a mechanism to prevent modification of an arbitrary subset of translation table entries from within the exception level that owns the translation tables, and introduces the following:

- The stage 1 Assured Translation property and the stage 2 AssuredOnly property.
- Stage 2 TopLevel checks.
- Read-Check-Write instructions, RCW* and RCWS*.
- Read-Check-Write mask registers, RCWMASK_EL1 and RCWSMASK_EL1.
- The Protected attribute in stage 1 descriptors.

This feature is supported in AArch64 state only.

FEAT_THE is OPTIONAL from Armv8.8.

When FEAT_THE and FEAT_AA64EL2 are implemented, FEAT_S2PIE is implemented.

When FEAT_THE and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

If FEAT_THE is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_THE:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64PFR1_EL1.THE.

For more information, see:

- Translation Hardening Extension.
- Read-Check-Write.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.3 Armv9-A architecture extensions

The AArch32 state might optionally be implemented at EL0. The AArch32 state is not implemented at EL1, EL2, and EL3.

The implementation of FEAT_DoubleLock in an Armv9 implementation is prohibited.

An implementation of the Armv9-A architecture cannot include an ETM.

A2.3.1 The Armv9.0 architecture extension

The Armv9.0 architecture extension adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.0 compliant if all of the following apply:

- It is Armv8.5 compliant.
- It includes all of the Armv9.0 architectural features that are mandatory.

An Armv9.0 compliant implementation can additionally include:

- Armv9.0 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.1, subject only to those constraints that require that certain features be implemented together.

In an Armv9.0 implementation, the AArch32 state might optionally be implemented at EL0. The AArch32 state is not implemented at EL1, EL2, and EL3.

An implementation of Armv9.0 cannot implement FEAT_DoubleLock.

An implementation of the Armv9.0 architecture cannot include an ETM.

In an Armv9.0 implementation, if FEAT_FP and FEAT_AdvSIMD are implemented, the following features are implemented:

- FEAT_RDM.
- FEAT_FP16.
- FEAT_DotProd.
- FEAT_FHM.
- FEAT_FCMA.
- FEAT_JSCVT.
- FEAT_FlagM2.
- FEAT_FRINTTS.

FEAT_Armv9_Crypto, Armv9 Cryptographic Extension

The Armv9 Cryptographic Extension provides instructions for the acceleration of encryption and decryption. The presence of the Cryptographic Extension in an implementation is subject to export license controls.

This feature is supported in AArch64 state only.

FEAT_Armv9_Crypto is OPTIONAL.

If FEAT_Armv9_Crypto is implemented, then FEAT_Crypto is implemented.

If FEAT_Armv9_Crypto is implemented, then FEAT_PMULL, FEAT_AES, FEAT_SHA1, FEAT_SHA3, FEAT_SHA256, and FEAT_SHA512 are implemented.

When FEAT_Armv9_Crypto, FEAT_SVE and FEAT_SVE_SHA3 are implemented, FEAT_SVE_AES, and FEAT_SVE_PMULL128 are implemented.

When FEAT_Armv9_Crypto, FEAT_SVE and FEAT_SVE_AES are implemented, FEAT_SVE_PMULL128, and FEAT_SVE_SHA3 are implemented.

When FEAT_Armv9_Crypto, FEAT_SVE and FEAT_SVE_PMULL128 are implemented, FEAT_SVE_AES, and FEAT_SVE_SHA3 are implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- The Cryptographic Extension in AArch64 state.
- The Cryptographic Extension in AArch32 state.

FEAT_ETE, Embedded Trace Extension

FEAT_ETE provides a trace unit that records details about software control flow running on a PE, which can be used to aid debugging or optimizing. The trace unit provides filtering functionality to allow the targeting of the information to specific code regions or periods of operation.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

FEAT_ETE is OPTIONAL from Armv9.0.

If FEAT_ETE is implemented, then FEAT_TRBE is implemented.

If FEAT_ETE is implemented, then FEAT_TRF is implemented.

If FEAT_ETE is implemented, then FEAT_TRC_SR is implemented.

If FEAT_ETE is implemented, then FEAT_ETMv4 is not implemented.

When FEAT_ETE and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

For more information, see The Embedded Trace Extension.

FEAT_SVE2, Scalable Vector Extension version 2

The Scalable Vector Extension version 2 (SVE2) is a superset of SVE that incorporates functionality similar to Advanced SIMD, and other enhancements. In this Manual, unless stated otherwise, when SVE is used, the behavior also applies to SVE2. All Armv9-A systems that support standard operating systems with rich application environments also provide hardware support for SVE2 instructions.

This feature is supported in AArch64 state only.

FEAT_SVE2 is OPTIONAL from Armv9.0.

If FEAT_SVE2 is implemented, then FEAT_SVE is implemented.

The following field identifies the presence of FEAT_SVE2:

- ID_AA64ZFR0_EL1.SVEver.

For more information, see:

- FEAT_SVE.
- Data processing - SVE2.

FEAT_SVE_AES, Scalable Vector AES instructions

FEAT_SVE_AES provides the following scalable vector AES cryptographic instructions:

- AESD.
- AESE.
- AESIMC.
- AESMC.
- PMULLB.
- PMULLT.

This feature is supported in AArch64 state only.

FEAT_SVE_AES is OPTIONAL from Armv9.0.

If FEAT_SVE_AES is implemented, then FEAT_SVE2 or FEAT_SSVE_AES is implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SVE_AES is implemented, then FEAT_SVE_PMULL128 is implemented.

If FEAT_SVE_PMULL128 is implemented, then FEAT_SVE_AES is implemented.

If FEAT_SVE_AES is implemented, then FEAT_Armv9_Crypto is implemented.

The following field identifies the presence of FEAT_SVE_AES:

- ID_AA64ZFR0_EL1.AES.

FEAT_SVE_BitPerm, Scalable Vector Bit Permutes instructions

FEAT_SVE_BitPerm provides the following scalable vector bit permute instructions:

- BEXT.
- BDEP.
- BGRP.

This feature is supported in AArch64 state only.

FEAT_SVE_BitPerm is OPTIONAL from Armv9.0.

If FEAT_SVE_BitPerm is implemented, then FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_SVE_BitPerm:

- ID_AA64ZFR0_EL1.BitPerm.

For more information, see:

- 'Bit permutation'.

FEAT_SVE_PMULL128, SVE single-vector Advanced Encryption Standard and 128-bit polynomial multiply long instructions

FEAT_SVE_PMULL128 implements the SVE single-vector Advanced Encryption Standard and 128-bit destination element variants of PMULLB and PMULLT instructions, when the PE is not in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SVE_PMULL128 is OPTIONAL from Armv9.0.

If FEAT_SVE_PMULL128 is implemented, then FEAT_SVE2 or FEAT_SSVE_AES is implemented.

If FEAT_SVE_PMULL128 is implemented, then FEAT_SVE_AES is implemented.

The following field identifies the presence of FEAT_SVE_PMULL128:

- ID_AA64ZFR0_EL1.AES.

FEAT_SVE_SHA3, Scalable Vector SHA3 instructions

FEAT_SVE_SHA3 provides the following scalable vector SHA3 instruction:

- RAX1.

This feature is supported in AArch64 state only.

FEAT_SVE_SHA3 is OPTIONAL from Armv9.0.

If FEAT_SVE_SHA3 is implemented, then FEAT_SVE2 or FEAT_SME2p1 is implemented.

If FEAT_SVE_SHA3 is implemented, then FEAT_Armv9_Crypto is implemented.

The following field identifies the presence of FEAT_SVE_SHA3:

- ID_AA64ZFR0_EL1.SHA3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SVE_SM4, Scalable Vector SM4 instructions

FEAT_SVE_SM4 provides the following scalable vector SM4 instructions:

- SM4E.
- SM4EKEY.

This feature is supported in AArch64 state only.

FEAT_SVE_SM4 is OPTIONAL from Armv9.0.

If FEAT_SVE_SM4 is implemented, then FEAT_SVE2 is implemented.

If FEAT_SVE_SM4 is implemented, then FEAT_SM4 is implemented.

The following field identifies the presence of FEAT_SVE_SM4:

- ID_AA64ZFR0_EL1.SM4.

FEAT_TRBE, Trace Buffer Extension

FEAT_TRBE enables support for a Trace Buffer Unit within a PE. When the Trace Buffer Unit is enabled, program-flow trace generated by a trace unit is written directly to memory by the Trace Buffer Unit, rather than routing trace data to a trace sink.

This feature is supported in AArch64 state, and collects trace in both AArch64 and AArch32 states.

FEAT_TRBE is OPTIONAL from Armv9.0.

If FEAT_TRBE is implemented, then FEAT_TRF is implemented.

If FEAT_TRBE is implemented, then FEAT_ETE is implemented.

When FEAT_TRBE and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

The following field identifies the presence of FEAT_TRBE:

- ID_AA64DFR0_EL1.TraceBuffer.

For more information, see The Trace Buffer Extension.

A2.3.1.1 Features added to the Armv9.0 extension in later releases

- FEAT_IDTE3.
- FEAT_PCDPHINT.
- FEAT_UINJ.

A2.3.2 The Armv9.1 architecture extension

The Armv9.1 architecture extension is an extension to Armv9.0. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.1 compliant if all of the following apply:

- It is Armv8.6 compliant.
- It is Armv9.0 compliant.
- It includes all of the Armv9.1 architectural features that are mandatory.

An Armv9.1 compliant implementation can additionally include:

- Armv9.1 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.2, subject only to those constraints that require that certain features be implemented together.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ETEv1p1, Embedded Trace Extension

FEAT_ETEv1p1 extends FEAT_ETE to provide more flexibility for tracing Timestamp values.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

FEAT_ETEv1p1 is OPTIONAL from Armv9.0.

If FEAT_ETEv1p1 is implemented, then FEAT_ETE is implemented.

The following fields identify the presence of FEAT_ETEv1p1:

- TRCDEVARCH.REVISION.
- TRCDEVARCH.REVISION.

For more information, see The Embedded Trace Extension..

A2.3.3 The Armv9.2 architecture extension

The Armv9.2 architecture extension is an extension to Armv9.1. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.2 compliant if all of the following apply:

- It is Armv8.7 compliant.
- It is Armv9.1 compliant.
- It includes all of the Armv9.2 architectural features that are mandatory.

An Armv9.2 compliant implementation can additionally include:

- Armv9.2 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.3, subject only to those constraints that require that certain features be implemented together.

FEAT_BRBE, Branch Record Buffer Extension

FEAT_BRBE provides a Branch record buffer for capturing control path history.

This feature is supported in AArch64 state only.

FEAT_BRBE is OPTIONAL from Armv9.1.

The following field identifies the presence of FEAT_BRBE:

- ID_AA64DFR0_EL1.BRBE.

For more information, see The Branch Record Buffer Extension.

FEAT_ETEv1p2, Embedded Trace Extension

FEAT_ETEv1p2 extends FEAT_ETE to support FEAT_RME.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

FEAT_ETEv1p2 is OPTIONAL from Armv9.1.

If FEAT_ETEv1p2 is implemented, then FEAT_ETEv1p1 is implemented.

If FEAT_ETE and FEAT_RME are implemented, then FEAT_ETEv1p2 is implemented.

The following fields identify the presence of FEAT_ETEv1p2:

- TRCDEVARCH.REVISION.
- TRCDEVARCH.REVISION.

For more information, see The Embedded Trace Extension..

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_RME, Realm Management Extension

The Realm Management Extension (RME) is an extension to the Armv9 A-profile architecture.

RME introduces all of the following features:

- Two additional Security states, Root and Realm.
- Two additional physical address spaces, Root and Realm.
- The ability to dynamically transition memory granules between physical address spaces.
- Granule Protection Check mechanism.

FEAT_RME is one component of the Arm Confidential Compute Architecture (Arm CCA). Together with the other components of the Arm CCA, RME enables support for dynamic, attestable, and trusted execution environments (Realms) to be run on an Arm PE.

This feature is supported in AArch64 state only.

FEAT_RME is OPTIONAL from Armv9.1.

If FEAT_RME is implemented, then FEAT_AA64EL3, FEAT_AA64EL2, and FEAT_RNG or FEAT_RNG_TRAP are implemented.

When FEAT_RME and FEAT_AES or FEAT_SHA1 are implemented, FEAT_PMULL, FEAT_AES, FEAT_SHA3, FEAT_SHA256, and FEAT_SHA512 are implemented.

When FEAT_RME, FEAT_SVE and FEAT_AES or FEAT_SHA1 are implemented, FEAT_SVE_PMULL128, and FEAT_SVE_SHA3 are implemented.

When FEAT_ETE and FEAT_RME are implemented, FEAT_ETEv1p2 is implemented.

When FEAT_PMUv3 and FEAT_RME are implemented, FEAT_PMUv3p7 is implemented.

When FEAT_SPE and FEAT_RME are implemented, FEAT_SPEv1p2 is implemented.

When FEAT_MPAM and FEAT_RME are implemented, FEAT_MPAMv1p1 is implemented.

If FEAT_RME is implemented, then FEAT_PCSRv8 is not implemented.

The following field identifies the presence of FEAT_RME:

- ID_AA64PFR0_EL1.RME.

If the Activity Monitors Extension is implemented, then Arm strongly recommends that a PE that implements FEAT_RME also implements FEAT_AMUv1p1.

Arm recommends that a PE that implements FEAT_RME also implements all of the following:

- FEAT_VMID16.
- FEAT_HAFDBS.

For more information, see:

- The AArch64 System Level Programmers´ Model.
- The AArch64 Virtual Memory System Architecture.
- The Granule Protection Check Mechanism.

FEAT_SME, Scalable Matrix Extension

FEAT_SME introduces two AArch64 execution modes that can be enabled and disabled by application software:

- In ZA storage enabled mode, scalable, two-dimensional, architectural ZA tile storage becomes available and instructions are defined to load, store, extract, insert, and clear rows and columns of the ZA tiles.
- In Streaming SVE mode, the Effective SVE vector length changes to match the Effective ZA tile width, support for a substantial subset of the SVE2 instruction set is available, and, when ZA mode is also enabled, instructions are defined that accumulate the matrix outer product of two SVE vectors into a ZA tile.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in AArch64 state only.

FEAT_SME is OPTIONAL from Armv9.2.

If FEAT_SME is implemented, then FEAT_FCMA, FEAT_FP16, FEAT_BF16, and FEAT_FHM are implemented.

When FEAT_SME and FEAT_EL2 are implemented, FEAT_FGT and FEAT_HCX are implemented.

When FEAT_SME and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

The following fields identify the presence of FEAT_SME:

- ID_AA64PFR1_EL1.SME.
- ID_AA64SMFR0_EL1.F16F32.
- ID_AA64SMFR0_EL1.B16F32.
- ID_AA64SMFR0_EL1.F32F32.
- ID_AA64SMFR0_EL1.I8I32.

If FEAT_SME is implemented, this does not imply that FEAT_SVE and FEAT_SVE2 are implemented when the PE is not in Streaming SVE mode.

For more information, see The Scalable Matrix Extension.

FEAT_SME_F64F64, Double-precision floating-point outer product instructions

FEAT_SME_F64F64 indicates SME support for instructions that accumulate into double-precision floating-point elements in the ZA array.

This feature is supported in AArch64 state only.

FEAT_SME_F64F64 is OPTIONAL from Armv9.2.

If FEAT_SME_F64F64 is implemented, then FEAT_SME is implemented.

The following field identifies the presence of FEAT_SME_F64F64:

- ID_AA64SMFR0_EL1.F64F64.

For more information, see The Scalable Matrix Extension.

FEAT_SME_FA64, Full A64 instruction set support in Streaming SVE mode

FEAT_SME_FA64 indicates support for execution of the full A64 instruction set in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SME_FA64 is OPTIONAL from Armv9.2.

If FEAT_SME_FA64 is implemented, then FEAT_SME is implemented.

If FEAT_SME_FA64 is implemented, then FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_SME_FA64:

- ID_AA64SMFR0_EL1.FA64.

For more information, see The Scalable Matrix Extension.

FEAT_SME_I16I64, 16-bit to 64-bit integer widening outer product instructions

FEAT_SME_I16I64 indicates SME support for instructions that accumulate into 64-bit integer elements in the ZA array.

This feature is supported in AArch64 state only.

FEAT_SME_I16I64 is OPTIONAL from Armv9.2.

If FEAT_SME_I16I64 is implemented, then FEAT_SME is implemented.

The following field identifies the presence of FEAT_SME_I16I64:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64SMFR0_EL1.I16I64.

For more information, see The Scalable Matrix Extension.

A2.3.3.1 Features added to the Armv9.2 extension in later releases

- FEAT_F8F16MM.
- FEAT_F8F32MM.
- FEAT_FAMINMAX.
- FEAT_FP8DOT2.
- FEAT_FP8DOT4.
- FEAT_FP8FMA.
- FEAT_FP8.
- FEAT_FPMR.
- FEAT_LS64WB.
- FEAT_LUT.
- FEAT_SME2p1.
- FEAT_SME_B16B16.
- FEAT_SME_F16F16.
- FEAT_SME_F8F16.
- FEAT_SME_F8F32.
- FEAT_SME_LUTv2.
- FEAT_SPE_SME.
- FEAT_SSVE_FP8DOT2.
- FEAT_SSVE_FP8DOT4.
- FEAT_SSVE_FP8FMA.
- FEAT_SVE2p1.
- FEAT_SVE_B16B16.
- FEAT_SVE_BFSCALE.
- FEAT_SVE_F16F32MM.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.3.4 The Armv9.3 architecture extension

The Armv9.3 architecture extension is an extension to Armv9.2. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.3 compliant if all of the following apply:

- It is Armv8.8 compliant.
- It is Armv9.2 compliant.
- It includes all of the Armv9.3 architectural features that are mandatory.

An Armv9.3 compliant implementation can additionally include:

- Armv9.3 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.4, subject only to those constraints that require that certain features be implemented together.

FEAT_BRBEv1p1, Branch Record Buffer Extension version 1.1

FEAT_BRBEv1p1 extends FEAT_BRBE to enable branch recording at EL3.

This feature is supported in AArch64 state only.

FEAT_BRBEv1p1 is OPTIONAL from Armv9.2.

In an Armv9.3 implementation, if FEAT_BRBE is implemented, FEAT_BRBEv1p1 is implemented.

If FEAT_BRBEv1p1 is implemented, then FEAT_BRBE is implemented.

The following field identifies the presence of FEAT_BRBEv1p1:

- ID_AA64DFR0_EL1.BRBE.

For more information, see The Branch Record Buffer Extension.

FEAT_MEC, Memory Encryption Contexts

Memory Encryption Contexts (MEC) is an extension to the RME.

An existing RME enabled system uses a combination of isolation and external memory protection to guarantee privacy of the Realm Security state. Isolation between Realms is enforced by the Realm stage 2 translation tables.

FEAT_MEC introduces all of the following features:

- Memory encryption contexts are provided to all physical address spaces.
- Multiple memory encryption contexts are provided to the Realm physical address space for assignment to Realm virtual machines, with policy controlled by Realm EL2.
- The Non-secure, Secure, and Root, physical address spaces each have one encryption context.

This feature is supported in AArch64 state only.

FEAT_MEC is OPTIONAL from Armv9.2.

If FEAT_MEC is implemented, then FEAT_RME is implemented.

If FEAT_MEC is implemented, then FEAT_SCTLR2 is implemented.

If FEAT_MEC is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_MEC:

- ID_AA64MMFR3_EL1.MEC.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SME2, Scalable Matrix Extensions version 2

FEAT_SME2 is a superset of FEAT_SME that introduces the following:

- The ability to treat the SME ZA array as containing addressable groups of one-dimensional ZA array vectors, instead of two-dimensional ZA tiles.
- Multi-vector instructions that operate on groups of Z vector registers and ZA array vectors.
- A multi-vector predication mechanism for multi-vector load and store.
- A dedicated 512-bit lookup table register, ZT0, for data decompression.

This feature is supported in AArch64 state only.

FEAT_SME2 is OPTIONAL from Armv9.2.

If FEAT_SME2 is implemented, then FEAT_SME is implemented.

The following fields identify the presence of FEAT_SME2:

- ID_AA64SMFR0_EL1.SMEver.
- ID_AA64PFR1_EL1.SME.
- ID_AA64SMFR0_EL1.BI32I32.
- ID_AA64SMFR0_EL1.I16I32.

For more information, see The Scalable Matrix Extension.

A2.3.4.1 Features added to the Armv9.3 extension in later releases

- FEAT_LSFE.
- FEAT_MPAM_PE_BW_CTRL.
- FEAT_OCCMO.

A2.3.5 The Armv9.4 architecture extension

The Armv9.4 architecture extension is an extension to Armv9.3. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.4 compliant if all of the following apply:

- It is Armv8.9 compliant.
- It is Armv9.3 compliant.
- It includes all of the Armv9.4 architectural features that are mandatory.

An Armv9.4 compliant implementation can additionally include:

- Armv9.4 features that are optional.

FEAT_ABLE, Address Breakpoint Linking Extension

FEAT_ABLE introduces the capability to link a watchpoint to an address matching breakpoint.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ABLE is OPTIONAL from Armv9.3.

If FEAT_ABLE is implemented, then FEAT_BWE is implemented.

If FEAT_ABLE is implemented, then FEAT_Debugv8p9 is implemented.

The following fields identify the presence of FEAT_ABLE:

- ID_AA64DFR1_EL1.ABLE.
- EDDFR1.ABLE.

For more information, see About Breakpoint exceptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_BWE, Breakpoint and watchpoint enhancements

FEAT_BWE introduces the capability to define an included-range-based breakpoint and an excluded-range-based breakpoint.

This feature is supported in AArch64 state only.

FEAT_BWE is OPTIONAL from Armv9.3.

The following fields identify the presence of FEAT_BWE:

- ID_AA64DFR2_EL1.BWE.
- ID_AA64DFR1_EL1.ABLE.
- EDDFR1.ABLE.
- EDDFR2.BWE.

For more information, see Other usage constraints for Address breakpoints.

FEAT_CHK, Check Feature Status

FEAT_CHK introduces the CHKFEAT instruction, which allows software to detect when certain features are enabled.

A PE that is compliant with architectures from Armv8.0 to Armv9.3 is compliant with the behavior defined for this feature.

This feature is supported in AArch64 state only.

FEAT_CHK is OPTIONAL from Armv8.0.

FEAT_CHK is mandatory from Armv9.4.

For more information, see Detecting when FEAT_GCS is enabled.

FEAT_D128, 128-bit Translation Tables, 56 bit PA

FEAT_D128 introduces support for the VMSAv9-128 translation system, comprising the following:

- 128-bit translation table descriptors.
- Support for encoding up to 56-bit physical addresses in translation table descriptors.
- If FEAT_LVA or FEAT_LVA3 are implemented, support for translating up to 56-bit virtual addresses.
- TLBIP VA*, TLBIP RVA*, TLBIP IPA*, TLBIP RIPA* instructions that can take 128-bit inputs.

This feature is supported in AArch64 state only.

FEAT_D128 is OPTIONAL from Armv9.3.

If FEAT_D128 is implemented, then FEAT_SYSREG128 is implemented.

If FEAT_D128 is implemented, then FEAT_SYSINSTR128 is implemented.

If FEAT_D128 is implemented, then FEAT_LSE128 is implemented.

If FEAT_D128 is implemented, then FEAT_S1PIE is implemented.

When FEAT_D128 and FEAT_EL2 are implemented, FEAT_S2PIE is implemented.

If FEAT_D128 is implemented, then FEAT_AIE is implemented.

If FEAT_D128 is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_D128:

- ID_AA64MMFR3_EL1.D128.

For more information, see The AArch64 Virtual Memory System Architecture.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_EBEP, Exception-based Event Profiling

FEAT_EBEP provides support for reporting PMU counter overflows as PMU Profiling exceptions. This allows generation of higher quality profiles by eliminating the interrupt latency and jitter incurred outside the PE.

This feature is supported in both AArch64 and AArch32 states.

FEAT_EBEP is OPTIONAL from Armv9.3.

In an Armv9.3 implementation, if FEAT_PMUv3p9 is implemented, FEAT_EBEP is implemented.

When FEAT_EBEP and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

When FEAT_EBEP and FEAT_AA32EL0 are implemented, FEAT_Debugv8p9 is implemented.

The following field identifies the presence of FEAT_EBEP:

- ID_AA64DFR1_EL1.EBEP.

For more information, see Exception-based event profiling.

FEAT_ETEv1p3, Embedded Trace Extension version 1.3

FEAT_ETEv1p3 extends FEAT_ETE to support all of the following:

- The ETE External Debug Request, when FEAT_Debugv8p9 is implemented.
- The ETE Trace Output Enable, which is mandatory for FEAT_ETEv1p3 and OPTIONAL for FEAT_ETE.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

FEAT_ETEv1p3 is OPTIONAL from Armv9.3.

If FEAT_ETEv1p3 is implemented, then FEAT_ETEv1p2 is implemented.

The following fields identify the presence of FEAT_ETEv1p3:

- TRCDEVARCH.REVISION.
- TRCDEVARCH.REVISION.

For more information, see The Embedded Trace Extension.

FEAT_GCS, Guarded Control Stack Extension

FEAT_GCS introduces support for a Guarded Control Stack, an area of memory in which procedure return addresses and exception return addresses are stored and protected from modification.

This feature is supported in AArch64 state only.

FEAT_GCS is OPTIONAL from Armv9.3.

If FEAT_GCS is implemented, then FEAT_CHK is implemented.

If FEAT_GCS is implemented, then FEAT_S1PIE is implemented.

The following field identifies the presence of FEAT_GCS:

- ID_AA64PFR1_EL1.GCS.

For more information, see Guarded Control Stack.

FEAT_ITE, Instrumentation Trace Extension

FEAT_ITE provides all of the following to allow software to inject instrumentation information into the ETE trace stream:

- The TRCIT instruction, that injects the value of a general purpose register into the ETE trace stream.
- Instrumentation Packet that contains the value written by the TRCIT instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Controls that define the behavior of the TRCIT instruction.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ITE is OPTIONAL from Armv9.3.

If FEAT_ITE is implemented, then FEAT_TRF is implemented.

If FEAT_ITE is implemented, then FEAT_ETE is implemented.

If FEAT_ITE is implemented, then FEAT_TRBE is implemented.

When FEAT_ITE and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_ITE:

- ID_AA64DFR1_EL1.ITE.
- TRCIDR0.ITE.
- TRCIDR0.ITE.

For more information, see:

- Instrumentation extension.
- Instrumentation element.

FEAT_LSE128, 128-bit Atomics

FEAT_LSE128 introduces support for 128-bit atomic instructions.

This feature is supported in AArch64 state only.

FEAT_LSE128 is OPTIONAL from Armv9.3.

If FEAT_LSE128 is implemented, then FEAT_LSE is implemented.

The following field identifies the presence of FEAT_LSE128:

- ID_AA64ISAR0_EL1.Atomic.

For more information, see:

- Atomic integer memory operations.
- Swap operations.

FEAT_LVA3, 56-bit VA

FEAT_LVA3 introduces support for 56-bit virtual addresses.

This feature is supported in AArch64 state only.

FEAT_LVA3 is OPTIONAL from Armv9.3.

If FEAT_LVA3 is implemented, then FEAT_D128 is implemented.

If FEAT_LVA3 is implemented, then FEAT_LVA is implemented.

The following field identifies the presence of FEAT_LVA3:

- ID_AA64MMFR2_EL1.VARange.

For more information, see Supported virtual address ranges.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SEBEP, Synchronous Exception-based Event Profiling

FEAT_SEBEP creates configurations to generate synchronous and precise PMU Profiling exceptions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SEBEP is OPTIONAL from Armv9.3.

If FEAT_SEBEP is implemented, then FEAT_EBEP is implemented.

When FEAT_SEBEP and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

When FEAT_SEBEP and FEAT_AA32EL0 are implemented, FEAT_Debugv8p9 is implemented.

The following field identifies the presence of FEAT_SEBEP:

- ID_AA64DFR0_EL1.SEBEP.

For more information, see:

- Synchronous exception-based event profiling.
- Synchronous events.

FEAT_SME2p1, Scalable Matrix Extension version 2.1

The Scalable Matrix Extension version 2.1 (SME2.1) is a superset of SME2 that adds:

- Support for SVE RAX1 instructions in Streaming SVE mode if FEAT_SVE_SHA3 is implemented.
- SME MOVAZ instructions that move and zero ZA array vectors or ZA tile slices.
- SME ZERO instructions that zero ZA array vectors.
- SME LUTI2 and LUTI4 lookup table instructions to Z vectors with strided numbering.

In this Manual, unless stated otherwise, when SME is used, the behavior also applies to SME2.1.

This feature is supported in AArch64 state only.

FEAT_SME2p1 is OPTIONAL from Armv9.2.

In an Armv9.4 implementation, if FEAT_SME2 is implemented, FEAT_SME2p1 is implemented.

If FEAT_SME2p1 is implemented, then FEAT_SME2 is implemented.

If FEAT_SME and FEAT_SVE2p1 are implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SME2p1:

- ID_AA64SMFR0_EL1.SMEver.

For more information, see:

- Data processing - SME, SME2.
- The Scalable Matrix Extension.

FEAT_SME_B16B16, Non-widening BFloat16 to BFloat16 SME ZA-targeting arithmetic

FEAT_SME_B16B16 introduces SME ZA-targeting non-widening BFloat16 floating-point instructions to SME.

This feature is supported in AArch64 state only.

FEAT_SME_B16B16 is OPTIONAL from Armv9.2.

If FEAT_SME_B16B16 is implemented, then FEAT_SME2 is implemented.

If FEAT_SME_B16B16 is implemented, then FEAT_SVE_B16B16 is implemented.

The following field identifies the presence of FEAT_SME_B16B16:

- ID_AA64SMFR0_EL1.B16B16.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SME_F16F16, Non-widening half-precision FP16 to FP16 arithmetic for SME2

FEAT_SME_F16F16 introduces the SME2 half-precision to single-precision convert instructions and non-widening half-precision floating-point instructions.

This feature is supported in AArch64 state only.

FEAT_SME_F16F16 is OPTIONAL from Armv9.2.

If FEAT_SME_F16F16 is implemented, then FEAT_SME2 is implemented.

The following field identifies the presence of FEAT_SME_F16F16:

- ID_AA64SMFR0_EL1.F16F16.

For more information, see:

- Data processing - SME, SME2.
- The Scalable Matrix Extension.

FEAT_SVE2p1, Scalable Vector Extensions version 2.1

The Scalable Vector Extension version 2.1 (SVE2.1) is a superset of SVE2 that adds:

- SVE instructions in Non-streaming SVE mode, which were previously added by SME in Streaming SVE mode. These are:
  - Contiguous multi-vector load and store instructions.
  - Predicate-as-counter instructions.
  - General-purpose SVE instructions.
- Other relaxations and enhancements.

In this Manual, unless stated otherwise, when SVE is used, the behavior also applies to SVE2.1.

This feature is supported in AArch64 state only.

FEAT_SVE2p1 is OPTIONAL from Armv9.2.

In an Armv9.4 implementation, if FEAT_SVE2 is implemented, FEAT_SVE2p1 is implemented.

If FEAT_SVE2p1 is implemented, then FEAT_SVE2 is implemented.

If FEAT_SVE2 and FEAT_SME2p1 are implemented, then FEAT_SVE2p1 is implemented.

The following field identifies the presence of FEAT_SVE2p1:

- ID_AA64ZFR0_EL1.SVEver.

For more information, see:

- FEAT_SVE.
- Loads and stores - SME, SME2, SVE2p1.
- Data processing - SVE2.

FEAT_SVE_B16B16, Non-widening BFloat16 to BFloat16 arithmetic for SVE2 and SME2

FEAT_SVE_B16B16 introduces the following:

- If FEAT_SVE2 is implemented, the SVE non-widening BFloat16 instructions when the PE is not in Streaming SVE mode.
- If FEAT_SME2 is implemented, the SVE non-widening BFloat16 instructions when the PE is in Streaming SVE mode, and the SME Z-targeting multi-vector non-widening BFloat16 instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in AArch64 state only.

FEAT_SVE_B16B16 is OPTIONAL from Armv9.2.

If FEAT_SVE_B16B16 is implemented, then FEAT_SME2 or FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_SVE_B16B16:

- ID_AA64ZFR0_EL1.B16B16.

For more information, see:

- BFloat16 arithmetic.
- BFloat16 minimum/maximum.
- Clamp to minimum/maximum.

FEAT_SYSINSTR128, 128-bit System instructions

FEAT_SYSINSTR128 introduces support for IMPLEMENTATION DEFINED System instructions that can take 128-bit inputs.

This feature is supported in AArch64 state only.

FEAT_SYSINSTR128 is OPTIONAL from Armv9.3.

If FEAT_SYSINSTR128 is implemented, then FEAT_SCTLR2 is implemented.

If FEAT_SYSINSTR128 is implemented, then FEAT_D128 is implemented.

The following field identifies the presence of FEAT_SYSINSTR128:

- ID_AA64ISAR2_EL1.SYSINSTR_128.

For more information, see The A64 System Instruction Class.

FEAT_SYSREG128, 128-bit System registers

FEAT_SYSREG128 introduces the following support for 128-bit System registers:

- The MRRS instruction to move a 128-bit System register into a pair of 64-bit general-purpose registers.
- The MSRR instruction to move a pair of 64-bit general-purpose registers to a 128-bit System register.
- 128-bit formats of the following system registers:
- The Physical Address Register, PAR_EL1.
- The Read-Check-Write mask registers, RCWMASK_EL1 and RCWSMASK_EL1.
- The following translation table base address registers, TTBR0_EL1, TTBR0_EL2, TTBR1_EL1, TTBR1_EL2, VTTBR_EL2.

This feature is supported in AArch64 state only.

FEAT_SYSREG128 is OPTIONAL from Armv9.3.

If FEAT_SYSREG128 is implemented, then FEAT_SCTLR2 is implemented.

If FEAT_SYSREG128 is implemented, then FEAT_D128 is implemented.

The following field identifies the presence of FEAT_SYSREG128:

- ID_AA64ISAR2_EL1.SYSREG_128.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TRBE_EXT, Trace Buffer external mode

FEAT_TRBE_EXT allows an external debugger, as well as a self-hosted debugger, to use the Trace Buffer Unit. All of the following registers are introduced to determine the parameters of the implementation:

- TRBDEVARCH.
- TRBDEVID.
- TRBDEVID1.

This feature is supported in both AArch64 and AArch32 states.

FEAT_TRBE_EXT is OPTIONAL from Armv9.3.

If FEAT_TRBE_EXT is implemented, then FEAT_TRBE is implemented.

The following fields identify the presence of FEAT_TRBE_EXT:

- ID_AA64DFR0_EL1.ExtTrcBuff.
- EDDFR.TraceBuffer.
- EDDFR.ExtTrcBuff.

For more information, see Trace buffer External mode.

FEAT_TRBE_MPAM, Trace Buffer MPAM extensions

FEAT_TRBE_MPAM allows software to program the MPAM PARTID and PMG to use different MPAM values for trace data. TRBDEVID1.{PMG_MAX, PARTID_MAX} are used to determine the parameters of the MPAM implementation.

This feature is supported in both AArch64 and AArch32 states.

FEAT_TRBE_MPAM is OPTIONAL from Armv9.3.

If FEAT_TRBE_MPAM is implemented, then FEAT_TRBE_EXT is implemented.

If FEAT_TRBE_MPAM is implemented, then FEAT_MPAM is implemented.

When FEAT_TRBE_MPAM and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_TRBE_MPAM:

- TRBIDR_EL1.MPAM.
- TRBIDR_EL1.MPAM.

For more information, see External mode and FEAT_MPAM.

A2.3.5.1 Features added to the Armv9.4 extension in later releases

- FEAT_RME_GDI.
- FEAT_SME_MOP4.
- FEAT_SME_TMOP.
- FEAT_SSVE_BitPerm.
- FEAT_SSVE_FEXPA.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.3.6 The Armv9.5 architecture extension

The Armv9.5 architecture extension is an extension to Armv9.4. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.5 compliant if all of the following apply:

- It is Armv9.4 compliant.
- It includes all of the Armv9.5 architectural features that are mandatory.

An Armv9.5 compliant implementation can additionally include:

- Armv9.5 features that are optional.

FEAT_ASID2, Support for concurrent use of two ASIDs

FEAT_ASID2 introduces support for use of one ASID for each TTBR_ELx.

This feature is supported in AArch64 state only.

FEAT_ASID2 is OPTIONAL from Armv9.4.

FEAT_ASID2 is mandatory from Armv9.5.

If FEAT_ASID2 is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_ASID2:

- ID_AA64MMFR4_EL1.ASID2.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

FEAT_BWE2, Breakpoint and watchpoint enhancements 2

FEAT_BWE2 provides the capability to generate a Watchpoint exception or debug event when software accesses an address outside of one or more user-supplied address regions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_BWE2 is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_BWE is implemented, FEAT_BWE2 is implemented.

If FEAT_BWE2 is implemented, then FEAT_BWE is implemented.

The following fields identify the presence of FEAT_BWE2:

- ID_AA64DFR2_EL1.BWE.
- EDDFR2.BWE.

For more information, see:

- About Watchpoint exceptions.
- Exception syndrome information, fault address information, and preferred return address.
- Watchpoint data address comparisons.

FEAT_CPA, Instruction-only Checked Pointer Arithmetic

FEAT_CPA introduces support for Checked Pointer Arithmetic instructions.

This feature is supported in AArch64 state only.

FEAT_CPA is OPTIONAL from Armv9.4.

FEAT_CPA is mandatory from Armv9.5.

The following field identifies the presence of FEAT_CPA:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64ISAR3_EL1.CPA.

For more information, see Checked Pointer Arithmetic.

FEAT_CPA2, Checked Pointer Arithmetic

FEAT_CPA2 introduces support for enabling Checked Pointer Arithmetic. Checked Pointer Arithmetic detects and reports corruption of the following:

- The top 8 bits of a pointer.
- Multiplication overflows that lead to invalid array indices being computed.

This feature is supported in AArch64 state only.

FEAT_CPA2 is OPTIONAL from Armv9.4.

If FEAT_CPA2 is implemented, then FEAT_CPA is implemented.

If FEAT_CPA2 is implemented, then FEAT_SCTLR2 is implemented.

The following field identifies the presence of FEAT_CPA2:

- ID_AA64ISAR3_EL1.CPA.

For more information, see Checked Pointer Arithmetic.

FEAT_E3DSE, Delegated SError exception injection

FEAT_E3DSE provides an SError injection mechanism for EL3 called delegated SErrors.

This feature is supported in AArch64 state only.

FEAT_E3DSE is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_AA64EL3 is implemented, FEAT_E3DSE is implemented.

The following field identifies the presence of FEAT_E3DSE:

- ID_AA64MMFR4_EL1.E3DSE.

For more information, see Asynchronous exception types.

FEAT_ETS3, Enhanced Translation Synchronization

FEAT_ETS3 introduces support for enhanced memory access ordering requirements for translation table walks.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ETS3 is OPTIONAL from Armv8.0.

FEAT_ETS3 is mandatory from Armv9.5.

If FEAT_ETS3 is implemented, then FEAT_ETS2 is implemented.

The following fields identify the presence of FEAT_ETS3:

- ID_AA64MMFR1_EL1.ETS.
- ID_MMFR5_EL1.ETS.
- ID_MMFR5.ETS.

For more information, see:

- Ordering requirements defined by the formal concurrency model.
- Ordering of memory accesses from translation table walks.
- Ordering of translation table walks.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FAMINMAX, Floating-point maximum and minimum absolute value instructions

FEAT_FAMINMAX introduces the Advanced SIMD, SVE and SME `FAMAX` and `FAMIN` instructions that compute floating-point maximum and minimum absolute value.

FEAT_FAMINMAX is OPTIONAL from Armv9.2.

In an Armv9.5 implementation, if FEAT_FP is implemented, FEAT_FAMINMAX is implemented.

If FEAT_FAMINMAX is implemented, then FEAT_AdvSIMD, FEAT_SVE2, or FEAT_SME2 is implemented.

The following field identifies the presence of FEAT_FAMINMAX:

- ID_AA64ISAR3_EL1.FAMINMAX.

For more information, see:

- SIMD arithmetic.
- Floating-point minimum/maximum absolute value.
- Multi-vector minimum/maximum.

FEAT_FGWTE3, Fine-Grained Write Trap EL3

FEAT_FGWTE3 introduces traps for write accesses at EL3 to individual *_EL3 System registers. The traps are independent of existing controls. The bits in the control register FGWTE3_EL3 are sticky, and writes of 0 are ignored.

This feature is supported in AArch64 state only.

FEAT_FGWTE3 is OPTIONAL from Armv9.4.

If FEAT_FGWTE3 is implemented, then FEAT_EL3 is implemented.

The following field identifies the presence of FEAT_FGWTE3:

- ID_AA64MMFR4_EL1.FGWTE3.

For more information, see EL3 configurable instruction controls.

FEAT_FP8, FP8 convert instructions

FEAT_FP8 introduces the following:

- OFP8 formats E5M2 and E4M3 for FP8 instructions.
- Advanced SIMD: FP8 convert instructions and a floating point scaling `FSCALE` instruction.
- SME: FP8 convert instructions and floating point scaling `FSCALE` instructions.
- SVE: FP8 convert instructions.

FEAT_FP8 is OPTIONAL from Armv9.2.

If FEAT_FP8 is implemented, then FEAT_FPMR is implemented.

If FEAT_FP8 is implemented, then FEAT_FAMINMAX is implemented.

If FEAT_FP8 is implemented, then FEAT_LUT is implemented.

If FEAT_FP8 is implemented, then FEAT_BF16 is implemented.

If FEAT_FP8 is implemented, then FEAT_AdvSIMD, FEAT_SVE2, or FEAT_SME2 is implemented.

The following fields identify the presence of FEAT_FP8:

- ID_AA64FPFR0_EL1.F8CVT.
- ID_AA64FPFR0_EL1.F8E4M3.
- ID_AA64FPFR0_EL1.F8E5M2.

For more information, see:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FP8 floating-point formats.
- Summary of FP8 instruction behaviors.
- For Advanced SIMD, FP8 floating-point conversion.
- For SVE, FP8 floating-point conversions.
- For SME, FP8 floating-point conversions.
- The *OCP 8-bit Floating Point Specification (OFP8)*.

FEAT_FP8DOT2, FP8 2-way dot product to half-precision instructions

FEAT_FP8DOT2 introduces the Advanced SIMD FP8 to half-precision 2-way dot product instructions.

If FEAT_SVE2 is implemented, FEAT_FP8DOT2 also implements the SVE versions of these instructions.

Note

This feature does not enable execution of the SVE instructions in Streaming SVE mode.

FEAT_FP8DOT2 is OPTIONAL from Armv9.2.

If FEAT_FP8DOT2 is implemented, then FEAT_FP8DOT4 is implemented.

If FEAT_FP8DOT2 is implemented, then FEAT_AdvSIMD or FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_FP8DOT2:

- ID_AA64FPFR0_EL1.F8DP2.

For more information, see:

- Summary of FP8 instruction behaviors.
- For Advanced SIMD, FP8 floating-point dot product.
- For SVE, FP8 floating-point dot product.

FEAT_FP8DOT4, FP8 4-way dot product to single-precision instructions

FEAT_FP8DOT4 introduces the following:

- Advanced SIMD: FP8 to single-precision 4-way dot product instructions.
- SVE: SVE FP8 to single-precision 4-way dot product instructions.

Note

This feature does not enable execution of the SVE instructions in Streaming SVE mode.

FEAT_FP8DOT4 is OPTIONAL from Armv9.2.

If FEAT_FP8DOT4 is implemented, then FEAT_FP8FMA is implemented.

If FEAT_FP8DOT4 is implemented, then FEAT_AdvSIMD or FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_FP8DOT4:

- ID_AA64FPFR0_EL1.F8DP4.

For more information, see:

- Summary of FP8 instruction behaviors.
- For Advanced SIMD, FP8 floating-point dot product.
- For SVE, FP8 floating-point dot product.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FP8FMA, FP8 multiply-accumulate to half-precision and single-precision instructions

FEAT_FP8FMA introduces the following:

- Advanced SIMD: FP8 to half-precision and single-precision multiply-accumulate instructions.
- SVE: FP8 to half-precision and single-precision multiply-accumulate instructions.

Note

This feature does not enable execution of the SVE instructions in Streaming SVE mode.

FEAT_FP8FMA is OPTIONAL from Armv9.2.

If FEAT_FP8FMA is implemented, then FEAT_FP8 is implemented.

If FEAT_FP8FMA is implemented, then FEAT_AdvSIMD or FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_FP8FMA:

- ID_AA64FPFR0_EL1.F8FMA.

For more information, see:

- Summary of FP8 instruction behaviors.
- For Advanced SIMD, FP8 floating-point widening multiply-accumulate.
- For SVE, FP8 floating-point widening multiply-accumulate.

FEAT_FPMR, Floating-point Mode Register

FEAT_FPMR introduces the Special-purpose register FPMR.

FEAT_FPMR is OPTIONAL from Armv9.2.

If FEAT_FPMR is implemented, then FEAT_AdvSIMD is implemented.

When FEAT_FPMR and FEAT_EL2 are implemented, FEAT_FGT is implemented.

When FEAT_FPMR and FEAT_EL2 are implemented, FEAT_HCX is implemented.

The following field identifies the presence of FEAT_FPMR:

- ID_AA64PFR2_EL1.FPMR.

For more information, see Summary of FP8 instruction behaviors.

FEAT_HACDBS, Hardware accelerator for cleaning Dirty state

FEAT_HACDBS introduces the Hardware accelerator for cleaning Dirty state, which enhances the process of updating stage 2 descriptors from writable-dirty to writable-clean.

This feature is supported in AArch64 state only.

FEAT_HACDBS is OPTIONAL from Armv9.4.

If FEAT_HACDBS is implemented, then FEAT_HDBSS is implemented.

The following field identifies the presence of FEAT_HACDBS:

- ID_AA64MMFR4_EL1.HACDBS.

For more information, see Hardware accelerator for cleaning dirty state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_HDBSS, Hardware Dirty state tracking structure

FEAT_HDBSS introduces the Hardware Dirty state tracking structure, which enhances tracking the dirty state of stage 2 translation table descriptors.

This feature is supported in AArch64 state only.

FEAT_HDBSS is OPTIONAL from Armv9.4.

If FEAT_HDBSS is implemented, then FEAT_HAFDBS is implemented.

If FEAT_HDBSS is implemented, then FEAT_AA64EL2 is implemented.

The following field identifies the presence of FEAT_HDBSS:

- ID_AA64MMFR1_EL1.HAFDBS.

For more information, see Hardware dirty state tracking structure.

FEAT_LUT, Lookup table instructions with 2-bit and 4-bit indices

FEAT_LUT introduces the following lookup table instructions with 2-bit and 4-bit indices:

- The Advanced SIMD `LUTI2` and `LUTI4` instructions.
- If FEAT_SVE2 is implemented, the SVE `LUTI2` and `LUTI4` instructions when the PE is not in Streaming SVE mode.
- If FEAT_SME2 is implemented, the SVE `LUTI2` and `LUTI4` instructions when the PE is in Streaming SVE mode.

FEAT_LUT is OPTIONAL from Armv9.2.

In an Armv9.5 implementation, if FEAT_AdvSIMD is implemented, FEAT_LUT is implemented.

If FEAT_LUT is implemented, then FEAT_AdvSIMD, FEAT_SVE2, or FEAT_SME2 is implemented.

The following field identifies the presence of FEAT_LUT:

- ID_AA64ISAR2_EL1.LUT.

For more information, see:

- For Advanced SIMD, Lookup table using 2-bit or 4-bit indices.
- For SVE, Lookup table using 2-bit or 4-bit indices.

FEAT_PAuth_LR, Pointer authentication instructions that allow signing of LR using SP and PC as diversifiers

FEAT_PAuth_LR introduces instructions that allow signing of the 64-bit value in LR against the value of SP, and a PC-relative instruction address.

This feature is supported in AArch64 state only.

FEAT_PAuth_LR is OPTIONAL from Armv9.4.

If FEAT_PAuth_LR is implemented, then FEAT_FPACCOMBINE is implemented.

If FEAT_PAuth_LR is implemented, then FEAT_SCTLR2 is implemented.

When FEAT_PAuth_LR and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following fields identify the presence of FEAT_PAuth_LR:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PMUv3_SME, Performance Monitors extensions for SME

FEAT_PMUv3_SME supports filtering to control counting events in Streaming and Non-streaming SVE modes.

This feature is supported in AArch64 state only.

FEAT_PMUv3_SME is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_PMUv3 and FEAT_SME are implemented, FEAT_PMUv3_SME is implemented.

If FEAT_PMUv3_SME is implemented, then FEAT_PMUv3 is implemented.

If FEAT_PMUv3_SME is implemented, then FEAT_SME is implemented.

The following field identifies the presence of FEAT_PMUv3_SME:

- PMMIR_EL1.SME.

For more information, see Filtering by SVE Streaming mode.

FEAT_PMUv3_TH2, Performance Monitors event counter linking extension

FEAT_PMUv3_TH2 extends the Performance Monitors Extension to add the ability to program an event counter to count the logical combination of two events.

This feature is supported in both AArch64 and AArch32 states. The threshold condition controls are accessible only in AArch64 state. However, threshold conditions still apply in AArch32 state.

FEAT_PMUv3_TH2 is OPTIONAL from Armv9.4.

If FEAT_PMUv3_TH2 is implemented, then FEAT_PMUv3_TH is implemented.

If FEAT_PMUv3_TH2 is implemented, then FEAT_PMUv3_EDGE is implemented.

The following fields identify the presence of FEAT_PMUv3_TH2:

- PMMIR_EL1.EDGE.
- PMMIR.EDGE.

For more information, see Event threshold and edge counting.

FEAT_RME_GPC2, RME Granule Protection Check 2 Extension

FEAT_RME_GPC2 introduces two mechanisms to the GPC that enable memory isolation models:

- Non-secure Only GPI encoding.
- Physical Address Space Disable.

This feature is supported in AArch64 state only.

FEAT_RME_GPC2 is OPTIONAL from Armv9.4.

If FEAT_RME_GPC2 is implemented, then FEAT_RME is implemented.

The following field identifies the presence of FEAT_RME_GPC2:

- ID_AA64PFR0_EL1.RME.

For more information, see:

- GPC faults.
- GPI field encoding in GPT descriptors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SME_F8F16, SME2 ZA-targeting FP8 multiply-accumulate, dot product, and outer product to half-precision instructions

FEAT_SME_F8F16 introduces the following SME instructions:

- ZA-targeting FP8 to half-precision multiply-accumulate, dot product, and outer product instructions.
- ZA-targeting multi-vector non-widening half-precision FADD and FSUB instructions.

FEAT_SME_F8F16 is OPTIONAL from Armv9.2.

If FEAT_SME_F8F16 is implemented, then FEAT_SME_F8F32 is implemented.

The following field identifies the presence of FEAT_SME_F8F16:

- ID_AA64SMFR0_EL1.F8F16.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point widening multiply-accumulate.
- FP8 floating-point dot product.
- FP8 floating-point outer product.

FEAT_SME_F8F32, SME2 ZA-targeting FP8 multiply-accumulate, dot product, and outer product to single-precision instructions

FEAT_SME_F8F32 introduces the SME ZA-targeting FP8 to single-precision multiply-accumulate, dot product, and outer product instructions.

FEAT_SME_F8F32 is OPTIONAL from Armv9.2.

If FEAT_SME_F8F32 is implemented, then FEAT_SME2 is implemented.

If FEAT_SME_F8F32 is implemented, then FEAT_FP8 is implemented.

The following field identifies the presence of FEAT_SME_F8F32:

- ID_AA64SMFR0_EL1.F8F32.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point widening multiply-accumulate.
- FP8 floating-point dot product.
- FP8 floating-point outer product.

FEAT_SME_LUTv2, Lookup table instructions with 4-bit indices and 8-bit elements

FEAT_SME_LUTv2 introduces the following:

- The LUTI4 (four registers, 8-bit) instruction.
- The MOVT (vector to table) instruction.

FEAT_SME_LUTv2 is OPTIONAL from Armv9.2.

If FEAT_SME_LUTv2 is implemented, then FEAT_SME2 is implemented.

The following field identifies the presence of FEAT_SME_LUTv2:

- ID_AA64SMFR0_EL1.LUTv2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPE_ALTCLK, Statistical Profiling alternate clock domain extension

FEAT_SPE_ALTCLK supports collection of timing data using the Statistical Profiling Extension when profiling a PE that includes asynchronous accelerators, such as a Streaming Mode Compute Unit (SMCU).

This feature is supported in AArch64 state only.

FEAT_SPE_ALTCLK is OPTIONAL from Armv9.4.

If FEAT_SPE_ALTCLK is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPE_ALTCLK:

- PMSIDR_EL1.ALTCLK.

For more information, see Asynchronous operation.

FEAT_SPE_EFT, Statistical Profiling extended filtering by type

FEAT_SPE_EFT extends SPE filtering to support targeted profiling of floating-point and SIMD operations.

This feature is supported in AArch64 state only.

FEAT_SPE_EFT is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_SPE is implemented, FEAT_SPE_EFT is implemented.

If FEAT_SPE_EFT is implemented, then FEAT_SPE is implemented.

If FEAT_SPE_EFT is implemented, then FEAT_SPE_FPF is implemented.

The following field identifies the presence of FEAT_SPE_EFT:

- PMSIDR_EL1.EFT.

For more information, see Filtering by operation type.

FEAT_SPE_FPF, Statistical Profiling floating-point and SIMD flag extension

FEAT_SPE_FPF extends the profiling information recorded by the Statistical Profiling Extension to identify sampled scalar floating-point and Advanced SIMD instructions.

This feature is supported in AArch64 state only.

FEAT_SPE_FPF is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_SPE is implemented, FEAT_SPE_FPF is implemented.

If FEAT_SPE_FPF is implemented, then FEAT_SPE is implemented.

If FEAT_SPE_FPF is implemented, then FEAT_SPE_EFT is implemented.

The following field identifies the presence of FEAT_SPE_FPF:

- PMSIDR_EL1.FPF.

For more information, see Additional information for other operations.

FEAT_SPE_SME, Statistical Profiling extensions for SME

FEAT_SPE_SME provides support for profiling of software that uses SME instructions using the Statistical Profiling Extension.

This feature is supported in AArch64 state only.

FEAT_SPE_SME is OPTIONAL from Armv9.2.

In an Armv9.5 implementation, if FEAT_SPEv1p1 and FEAT_SME are implemented, FEAT_SPE_SME is implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SPE_SME is implemented, then FEAT_SPEv1p1 is implemented.

If FEAT_SPE_SME is implemented, then FEAT_SME is implemented.

The following field identifies the presence of FEAT_SPE_SME:

- PMSIDR_EL1.SME.

For more information, see Additional information for each profiled SVE and SME operation.

FEAT_SPMU2, System Performance Monitors Extension version 2

FEAT_SPMU2 provides the functionality to set multiple event counters in a System PMU to zero in a single operation.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SPMU2 is OPTIONAL from Armv9.4.

If FEAT_SPMU and v9Ap5 are implemented, then FEAT_SPMU2 is implemented.

If FEAT_SPMU2 is implemented, then FEAT_SPMU is implemented.

The following field identifies the presence of FEAT_SPMU2:

- ID_AA64DFR1_EL1.SPMU.

For more information, see Zeroing System PMU counters.

FEAT_SSVE_FP8DOT2, SVE FP8 2-way dot product to half-precision instructions in Streaming SVE mode

FEAT_SSVE_FP8DOT2 enables execution of SVE FP8 to half-precision 2-way dot product instructions in Streaming SVE mode.

FEAT_SSVE_FP8DOT2 is OPTIONAL from Armv9.2.

If FEAT_SSVE_FP8DOT2 is implemented, then FEAT_SSVE_FP8DOT4 is implemented.

If FEAT_SME2 and FEAT_FP8DOT2 are implemented, then FEAT_SSVE_FP8DOT2 is implemented.

The following field identifies the presence of FEAT_SSVE_FP8DOT2:

- ID_AA64SMFR0_EL1.SF8DP2.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point dot product.

FEAT_SSVE_FP8DOT4, SVE2 FP8 4-way dot product to single-precision instructions in Streaming SVE mode

FEAT_SSVE_FP8DOT4 enables execution of SVE FP8 to single-precision 4-way dot product instructions in Streaming SVE mode.

FEAT_SSVE_FP8DOT4 is OPTIONAL from Armv9.2.

If FEAT_SSVE_FP8DOT4 is implemented, then FEAT_SSVE_FP8FMA is implemented.

If FEAT_SME2 and FEAT_FP8DOT4 are implemented, then FEAT_SSVE_FP8DOT4 is implemented.

The following field identifies the presence of FEAT_SSVE_FP8DOT4:

- ID_AA64SMFR0_EL1.SF8DP4.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point dot product.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SSVE_FP8FMA, SVE2 FP8 multiply-accumulate to half-precision and single-precision instructions in Streaming SVE mode

FEAT_SSVE_FP8FMA enables execution of SVE FP8 to half-precision and single-precision multiply-accumulate instructions in Streaming SVE mode.

FEAT_SSVE_FP8FMA is OPTIONAL from Armv9.2.

If FEAT_SSVE_FP8FMA is implemented, then FEAT_FP8 is implemented.

If FEAT_SSVE_FP8FMA is implemented, then FEAT_SME2 is implemented.

If FEAT_SME2 and FEAT_FP8FMA are implemented, then FEAT_SSVE_FP8FMA is implemented.

The following field identifies the presence of FEAT_SSVE_FP8FMA:

- ID_AA64SMFR0_EL1.SF8FMA.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point widening multiply-accumulate.

FEAT_STEP2, Enhanced Software Step Extension

FEAT_STEP2 introduces support for a debugger to control the instruction executed by the PE when single-stepping, without modifying the instruction in memory.

This feature is supported in both AArch64 and AArch32 states.

FEAT_STEP2 is OPTIONAL from Armv9.4.

FEAT_STEP2 is mandatory from Armv9.5.

When FEAT_STEP2 and FEAT_EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_STEP2:

- ID_AA64DFR2_EL1.STEP.

For more information, see Behavior in the active-not-pending state.

FEAT_TLBIW, TLBI VMALL for Dirty state

FEAT_TLBIW provides the following TLBI instructions that remove the stage 2 dirty state from TLB entries:

- TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS.
- TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS.
- TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS.

This feature is supported in AArch64 state only.

FEAT_TLBIW is OPTIONAL from Armv9.4.

If FEAT_TLBIW is implemented, then FEAT_AA64EL2 is implemented.

The following field identifies the presence of FEAT_TLBIW:

- ID_AA64ISAR3_EL1.TLBIW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.3.7 The Armv9.6 architecture extension

The Armv9.6 architecture extension is an extension to Armv9.5. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.6 compliant if all of the following apply:

- It is Armv9.5 compliant.
- It includes all of the Armv9.6 architectural features that are mandatory.

An Armv9.6 compliant implementation can additionally include:

- Armv9.6 features that are optional.

FEAT_AMU_EXTACR, Activity Monitors External Control Register

FEAT_AMU_EXTACR implements a control register to enable access to the Activity Monitors External registers.

- If FEAT_RME is implemented, the register AMROOTCR is implemented to control Root, Realm, Secure and Non-secure accesses to the external AMU registers.
- If FEAT_RME is not implemented, the register AMSCR is implemented to control Secure and Non-secure accesses to the external AMU registers.

FEAT_AMU_EXTACR is OPTIONAL from Armv8.4.

If FEAT_AMU_EXTACR is implemented, then FEAT_AMU_EXT is implemented.

For more information, see The Activity Monitors Extension.

FEAT_CMPBR, Compare and Branch instructions

FEAT_CMPBR introduces the A64 base compare and branch instructions.

This feature is supported in AArch64 state only.

FEAT_CMPBR is OPTIONAL from Armv9.5.

FEAT_CMPBR is mandatory from Armv9.6.

The following field identifies the presence of FEAT_CMPBR:

- ID_AA64ISAR2_EL1.CSSC.

For more information, see:

- ‘Conditional branch’.
- ‘Concurrent modification and execution of instructions’.

FEAT_F8F16MM, 8-bit floating-point matrix multiply-accumulate to half-precision

FEAT_F8F16MM introduces the Advanced SIMD 8-bit floating-point matrix multiply-accumulate to half-precision instruction.

If FEAT_SVE2 is implemented, FEAT_F8F16MM implements the SVE 8-bit floating-point matrix multiply-accumulate to half-precision instruction, when the PE is not in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_F8F16MM is OPTIONAL from Armv9.2.

If FEAT_F8F16MM is implemented, then FEAT_F8F32MM is implemented.

If FEAT_F8F16MM is implemented, then FEAT_FP8DOT2 is implemented.

The following field identifies the presence of FEAT_F8F16MM:

- ID_AA64FPFR0_EL1.F8MM4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- For Advanced SIMD: 'FP8 matrix multiply-accumulate'.
- For SVE: 'FP8 floating-point matrix multiply-accumulate'.

FEAT_F8F32MM, 8-bit floating-point matrix multiply-accumulate to single-precision

FEAT_F8F32MM introduces the Advanced SIMD 8-bit floating-point matrix multiply-accumulate to single-precision instruction.

If FEAT_SVE2 is implemented, FEAT_F8F32MM implements the SVE 8-bit floating-point matrix multiply-accumulate to single-precision instruction, when the PE is not in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_F8F32MM is OPTIONAL from Armv9.2.

If FEAT_F8F32MM is implemented, then FEAT_FP8DOT4 is implemented.

The following field identifies the presence of FEAT_F8F32MM:

- ID_AA64FPFR0_EL1.F8MM8.

For more information, see:

- For Advanced SIMD: 'FP8 matrix multiply-accumulate'.
- For SVE: 'FP8 floating-point matrix multiply-accumulate'.

FEAT_FPRCVT, Floating-Point to/from Integer in Scalar FP register

FEAT_FPRCVT introduces A64 base floating-point to integer and integer to floating-point convert instructions with only scalar SIMD&FP register operands and results, and with different input and output register sizes.

FEAT_FPRCVT implements the Advanced SIMD scalar floating-point to integer and integer to floating-point convert instructions with only scalar SIMD&FP register operands and results, and with different and same input and output register sizes, when the PE is in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_FPRCVT is OPTIONAL from Armv9.5.

If FEAT_FP and v9Ap6 are implemented, then FEAT_FPRCVT is implemented.

If FEAT_FPRCVT is implemented, then FEAT_FP is implemented.

The following field identifies the presence of FEAT_FPRCVT:

- ID_AA64ISAR3_EL1.FPRCVT.

For more information, see:

- 'Floating-point conversion'.

FEAT_IDTE3, Trapping ID register accesses to EL3

FEAT_IDTE3 introduces support for trapping ID register accesses to EL3.

This feature is supported in AArch64 state only.

In an Armv9.6 implementation, if FEAT_EL3 is implemented, FEAT_IDTE3 is implemented.

FEAT_IDTE3 is OPTIONAL from Armv9.0.

The following field identifies the presence of FEAT_IDTE3:

- ID_AA64MMFR2_EL1.IDS.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LS64WB, LS64 for Write-back cacheable memory

FEAT_LS64WB provides support for large single-copy atomic load and store instructions for accesses to Conventional memory to enable scalability of atomic message passing without relying on separate flag and data accesses.

This feature is supported in AArch64 state only.

FEAT_LS64WB is OPTIONAL from Armv9.2.

If FEAT_LS64WB is implemented, then FEAT_LS64_ACCDATA is implemented.

The following field identifies the presence of FEAT_LS64WB:

- ID_AA64ISAR1_EL1.LS64.

FEAT_LSFE, Large System Float Extension

FEAT_LSFE implements A64 base Atomic floating-point in-memory instructions.

This feature is supported in AArch64 state only.

FEAT_LSFE is OPTIONAL from Armv9.3.

If FEAT_LSFE is implemented, then FEAT_FP is implemented.

The following field identifies the presence of FEAT_LSFE:

- ID_AA64ISAR3_EL1.LSFE.

For more information, see:

- ‘Atomic floating-point memory operations’.

FEAT_LSUI, Unprivileged Load Store

FEAT_LSUI introduces unprivileged variants of load and store instructions so that clearing PSTATE.PAN is never required in privileged software.

This feature is supported in AArch64 state only.

FEAT_LSUI is OPTIONAL from Armv9.5.

FEAT_LSUI is mandatory from Armv9.6.

The following field identifies the presence of FEAT_LSUI:

- ID_AA64ISAR3_EL1.LSUI.

FEAT_MPAM_MSC_DCTRL, MPAM Default Resource Control

FEAT_MPAM_MSC_DCTRL introduces support in an MSC for a default resource control configuration, which is applied to requests with PARTIDs outside the MSC’s supported range, or with PARTIDs for which there is no configuration set.

This feature is supported in AArch64 state only.

FEAT_MPAM_MSC_DCTRL is OPTIONAL from Armv9.5.

If FEAT_MPAM_MSC_DCTRL is implemented, then FEAT_MPAMv1p1 or FEAT_MPAMv0p1 is implemented.

The following field identifies the presence of FEAT_MPAM_MSC_DCTRL:

- MPAMF_IDR.HAS_DEFAULT_PARTID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_MPAM_MSC_DOMAINS, MPAM Domains Identifier Translation

FEAT_MPAM_MSC_DOMAINS introduces support for MSCs to manipulate the MPAM information bundle by applying PARTID ingress translation, egress translation, or both.

This feature is supported in AArch64 state only.

FEAT_MPAM_MSC_DOMAINS is OPTIONAL from Armv9.5.

If FEAT_MPAM_MSC_DOMAINS is implemented, then FEAT_MPAMv1p1 or FEAT_MPAMv0p1 is implemented.

The following fields identify the presence of FEAT_MPAM_MSC_DOMAINS:

- MPAMF_IDR.HAS_IN_TL.
- MPAMF_IDR.HAS_OUT_TL.

FEAT_MPAM_PE_BW_CTRL, MPAM PE-side Bandwidth Controls

FEAT_MPAM_PE_BW_CTRL introduces support for PE-side bandwidth controls.

This feature is supported in AArch64 state only.

FEAT_MPAM_PE_BW_CTRL is OPTIONAL from Armv9.3.

If FEAT_MPAM_PE_BW_CTRL is implemented, then FEAT_MPAMv1p1 or FEAT_MPAMv0p1 is implemented.

The following field identifies the presence of FEAT_MPAM_PE_BW_CTRL:

- MPAMIDR_EL1.HAS_BW_CTRL.

FEAT_NV2p1, Enhanced nested virtualization support

FEAT NV2p1 further supports nested virtualization by ensuring that control bits in EL1 registers are stateful if the corresponding bits in the corresponding EL2 registers are stateful. This is to prevent loss of the Guest hypervisor's state, and to permit the Host hypervisor to correctly emulate the EL2 environment.

This feature is supported in AArch64 state only.

FEAT_NV2p1 is OPTIONAL from Armv8.4.

In an Armv9.6 implementation, if FEAT_NV is implemented, FEAT_NV2p1 is implemented.

If FEAT_NV2p1 is implemented, then FEAT_NV2 is implemented.

The following field identifies the presence of FEAT_NV2p1:

- ID_AA64MMFR4_EL1.NV_frac.

FEAT_OCCMO, Outer Cacheable Cache Maintenance Operation

FEAT_OCCMO introduces the DC CIVAOC system instruction that provides software with a mechanism to publish writes to the Outer cache level.

This feature is supported in AArch64 state only.

FEAT_OCCMO is OPTIONAL from Armv9.3.

FEAT_OCCMO is mandatory from Armv9.6.

The following field identifies the presence of FEAT_OCCMO:

- ID_AA64ISAR3_EL1.OCCMO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PCDPHINT, Producer-Consumer Data Placement Hints

FEAT_PCDPHINT provides hint instructions to indicate each of the following:

- A store in the current execution thread is generating data at a specific location, which a thread of execution on one or more other observers is waiting on.
- The thread of execution on the current PE will read a location that may not yet have been written with the value to be consumed.

This feature is supported in AArch64 state only.

FEAT_PCDPHINT is OPTIONAL from Armv9.0.

The following field identifies the presence of FEAT_PCDPHINT:

- ID_AA64ISAR2_EL1.PCDPHINT.

FEAT_PMUv3_EXTPMN, Reserving PMU event counters for external agents

FEAT_PMUv3_EXTPMN provides external agents with a mechanism to reserve event counters for external use.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3_EXTPMN is OPTIONAL from Armv9.5.

If FEAT_PMUv3_EXTPMN is implemented, then FEAT_PMUv3_EXT is implemented.

If FEAT_PMUv3_EXTPMN is implemented, then FEAT_FGT is implemented.

When FEAT_PMUv3_EXTPMN and FEAT_EL2 are implemented, FEAT_HPMN0 is implemented.

The following field identifies the presence of FEAT_PMUv3_EXTPMN:

- PMDEVID.EXTPMN.

FEAT_PoPS, Point of Physical Storage

FEAT_PoPS defines the Point of Physical Storage (PoPS) and introduces System instructions to clean and invalidate to the PoPS.

This feature is supported in AArch64 state only.

FEAT_PoPS is OPTIONAL from Armv9.5.

The following field identifies the presence of FEAT_PoPS:

- ID_AA64MMFR4_EL1.PoPS.

For more information, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

FEAT_RME_GDI, RME Granular Data Isolation extension

FEAT_RME_GDI introduces additional GPT GPI encodings to enable memory isolation of non-PE data flows from the PEs, within an RME system.

This feature is supported in AArch64 state only.

FEAT_RME_GDI is OPTIONAL from Armv9.4.

If FEAT_RME_GDI is implemented, then FEAT_RME is implemented.

If FEAT_RME_GDI is implemented, then FEAT_RME_GPC2 is implemented.

The following field identifies the presence of FEAT_RME_GDI:

- ID_AA64MMFR4_EL1.RMEGDI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_RME_GPC3, RME Granule Protection Check 3 Extension

FEAT_RME_GPC3 introduces a method for defining a set of windows in the memory map for which Granule Protection Table lookups are skipped, and instead applies a set of default settings associated with the window.

FEAT_RME_GPC3 introduces the APAS system instruction, which provides an architected mechanism for directly programming the PA space association for regions of memory that support memory-side dynamic PA space association.

This feature is supported in AArch64 state only.

FEAT_RME_GPC3 is OPTIONAL from Armv9.5.

If FEAT_RME_GPC3 is implemented, then FEAT_RME is implemented.

If FEAT_RME_GPC3 is implemented, then FEAT_RME_GPC2 is implemented.

The following field identifies the presence of FEAT_RME_GPC3:

- ID_AA64PFR0_EL1.RME.

FEAT_SME2p2, Scalable Matrix Extension version 2.2

The Scalable Matrix Extension version 2.2 (SME2.2) is a superset of SME2.1 that adds:

- SME multi-vector Z-targeting non-widening floating-point FMUL instruction.
- Quarter-tile outer product instructions.
- Structured sparsity outer product instructions.
- Support for SVE2.2 instructions in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SME2p2 is OPTIONAL from Armv9.5.

In an Armv9.6 implementation, if FEAT_SME is implemented, FEAT_SME2p2 is implemented.

If FEAT_SME2p2 is implemented, then FEAT_SME2p1 is implemented.

If FEAT_SME2p2 is implemented, then FEAT_SME_MOP4 is implemented.

If FEAT_SME2p2 is implemented, then FEAT_SME_TMOP is implemented.

If FEAT_SME2p2 is implemented, then FEAT_SSVE_FEXPA is implemented.

If FEAT_SME and FEAT_SVE2p2 are implemented, then FEAT_SME2p2 is implemented.

The following field identifies the presence of FEAT_SME2p2:

- ID_AA64SMFR0_EL1.SMEver.

For more information, see:

- The Scalable Matrix Extension.
- ‘Multi-vector multiply’.
- ‘Quarter-tile outer product’.
- ‘Structured sparsity outer product’.

FEAT_SME_MOP4, Quarter-tile outer product instructions

FEAT_SME_MOP4 introduces the SME Quarter-tile outer product instructions

FEAT_SME_MOP4 is OPTIONAL from Armv9.4.

If FEAT_SME_MOP4 is implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SME_MOP4:

- ID_AA64SMFR0_EL1.SMOP4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- ‘Quarter-tile outer product’.

FEAT_SME_TMOP, Structured sparsity outer product instructions

FEAT_SME_TMOP introduces the SME Structured sparsity outer product instructions.

FEAT_SME_TMOP is OPTIONAL from Armv9.4.

If FEAT_SME_TMOP is implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SME_TMOP:

- ID_AA64SMFR0_EL1.STMOP.

For more information, see:

- ‘Structured sparsity outer product’.

FEAT_SPE_EXC, SPE Profiling exception extension

FEAT_SPE_EXC provides support for reporting Profiling Buffer management events as SPE Profiling exceptions.

This feature is supported in AArch64 state only.

FEAT_SPE_EXC is OPTIONAL from Armv9.5.

If FEAT_SPE_EXC is implemented, then FEAT_SPEv1p5 is implemented.

The following field identifies the presence of FEAT_SPE_EXC:

- ID_AA64DFR2_EL1.SPE_EXC.

FEAT_SPE_nVM, Statistical Profiling physical addressing mode extension

FEAT_SPE_nVM provides support for defining the Profiling Buffer using physical addresses or intermediate physical addresses.

This feature is supported in AArch64 state only.

FEAT_SPE_nVM is OPTIONAL from Armv9.5.

If FEAT_SPE_nVM is implemented, then FEAT_SPE is implemented.

When FEAT_SPE_nVM and FEAT_EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_SPE_nVM:

- ID_AA64DFR2_EL1.SPE_nVM.

FEAT_SPEv1p5, Statistical Profiling Extension version 1.5

FEAT_SPEv1p5 introduces the following to the Statistical Profiling Extension:

- If EL2 and FEAT_FGT are implemented, a fine-grained trap on the `PSB CSYNC` instruction.
- The SPE Profiling exception extension, FEAT_SPE_EXC.
- The Statistical Profiling physical addressing mode extension, FEAT_SPE_nVM.

This feature is supported in AArch64 state only.

FEAT_SPEv1p5 is OPTIONAL from Armv9.5.

In an Armv9.6 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p5 is implemented.

If FEAT_SPEv1p5 is implemented, then FEAT_SPEv1p4 is implemented.

If FEAT_SPEv1p5 is implemented, then FEAT_SPE_CRR is implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SPEv1p5 is implemented, then FEAT_SPE_EXC is implemented.

If FEAT_SPEv1p5 is implemented, then FEAT_SPE_nVM is implemented.

The following field identifies the presence of FEAT_SPEv1p5:

- ID_AA64DFR0_EL1.PMSVer.

FEAT_SRMASK, Bitwise System Register Write Masks

FEAT_SRMASK introduces aliases and bitwise write masks for EL1 control registers to reduce trapping of EL1 System register accesses to EL2. The equivalent bitwise write masks are also present for EL2.

This feature is supported in AArch64 state only.

FEAT_SRMASK is OPTIONAL from Armv9.5.

FEAT_SRMASK is mandatory from Armv9.6.

If FEAT_SRMASK is implemented, then FEAT_E2H0 is not implemented.

The following field identifies the presence of FEAT_SRMASK:

- ID_AA64MMFR4_EL1.SRMASK.

FEAT_SSVE_AES, Streaming SVE Mode Advanced Encryption Standard and 128-bit polynomial multiply long instructions

FEAT_SSVE_AES implements the SVE AES and 128-bit polynomial multiply long instructions, identified as implemented by ID_AA64ZFR0_EL1.AES field, when the PE is in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SSVE_AES is OPTIONAL from Armv9.5.

If FEAT_SSVE_AES is implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SSVE_AES:

- ID_AA64SMFR0_EL1.AES.

For more information, see:

- ‘AES-128 instructions’.
- ‘Polynomial arithmetic’.

FEAT_SSVE_BitPerm, Streaming Scalable Vector Bit Permutes instructions

FEAT_SSVE_BitPerm implements the SVE2 bit permute instructions, identified as implemented by ID_AA64ZFR0_EL1.BitPerm, when the PE is in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SSVE_BitPerm is OPTIONAL from Armv9.4.

If FEAT_SSVE_BitPerm is implemented, then FEAT_SME2p1 is implemented.

When FEAT_SSVE_BitPerm and FEAT_SVE2 is implemented, FEAT_SVE_BitPerm is implemented.

The following field identifies the presence of FEAT_SSVE_BitPerm:

- ID_AA64SMFR0_EL1.SBitPerm.

For more information, see:

- ‘Bit permutation’.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SSVE_FEXPA, Streaming FEXPA instruction

FEAT_SSVE_FEXPA implements the SVE FEXPA instruction when the PE is in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SSVE_FEXPA is OPTIONAL from Armv9.4.

If FEAT_SSVE_FEXPA is implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SSVE_FEXPA:

- ID_AA64SMFR0_EL1.SFEXPA.

FEAT_SVE2p2, Scalable Vector Extensions version 2.2

The Scalable Vector Extension version 2.2 (SVE2.2) is a superset of SVE2.1 that adds:

- Floating-point round to integral floating-point value instructions.
- Scalar index of first/last true predicate element.
- Zeroing-predication variant to all SVE unary instructions.
- Additional variants of `COMPACT` and `EXPAND` instructions.

This feature is supported in AArch64 state only.

FEAT_SVE2p2 is OPTIONAL from Armv9.5.

In an Armv9.6 implementation, if FEAT_SVE2 is implemented, FEAT_SVE2p2 is implemented.

If FEAT_SVE2p2 is implemented, then FEAT_SVE2p1 is implemented.

If FEAT_SVE2 and FEAT_SME2p2 are implemented, then FEAT_SVE2p2 is implemented.

The following field identifies the presence of FEAT_SVE2p2:

- ID_AA64ZFR0_EL1.SVEver.

For more information, see:

- ‘Predicate element index’.
- ‘Floating-point round’.
- ‘Vector permutation’.

FEAT_SVE_AES2, SVE multi-vector Advanced Encryption Standard and 128-bit polynomial multiply long instructions

FEAT_SVE_AES2 implements the SVE multi-vector Advanced Encryption Standard and 128-bit destination element polynomial multiply long instructions, when the PE is not in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SVE_AES2 is OPTIONAL from Armv9.5.

If FEAT_SVE_AES2 is implemented, then FEAT_SVE_PMULL128 is implemented.

If FEAT_SVE_AES2 is implemented, then FEAT_SVE2p1 or FEAT_SSVE_AES is implemented.

The following field identifies the presence of FEAT_SVE_AES2:

- ID_AA64ZFR0_EL1.AES.

For more information, see:

- ‘AES-128 instructions’.
- ‘Polynomial arithmetic’.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SVE_BFSCALE, BFloat16 Floating-Point Adjust Exponent

FEAT_SVE_BFSCALE introduces the following:

- If FEAT_SVE2 is implemented, the SVE `BFSCALE` instruction when the PE is not in Streaming SVE mode.
- If FEAT_SME2 is implemented, the SVE `BFSCALE` instruction when the PE is in Streaming SVE mode, and the SME multi-vector Z-targeting BFloat16 scaling instructions, `BFSCALE` and `BFMUL`.

This feature is supported in AArch64 state only.

FEAT_SVE_BFSCALE is OPTIONAL from Armv9.2.

If FEAT_SVE_BFSCALE is implemented, then FEAT_SVE_B16B16 is implemented.

The following field identifies the presence of FEAT_SVE_BFSCALE:

- ID_AA64ZFR0_EL1.B16B16.

For more information, see:

- For SVE: 'Floating-point adjust exponent'.
- For SME: 'Floating-point adjust exponent'.

FEAT_SVE_F16F32MM, SVE Half-precision floating-point matrix multiply-accumulate to single-precision

FEAT_SVE_F16F32MM introduces the SVE half-precision floating-point matrix multiply-accumulate to single-precision instruction.

This feature is supported in AArch64 state only.

FEAT_SVE_F16F32MM is OPTIONAL from Armv9.2.

If FEAT_SVE_F16F32MM is implemented, then FEAT_SVE2p1 is implemented.

The following field identifies the presence of FEAT_SVE_F16F32MM:

- ID_AA64ZFR0_EL1.F16MM.

For more information, see:

- 'Floating-point matrix multiply-accumulate'.

FEAT_TRBE_EXC, Trace Buffer Profiling exception extension

FEAT_TRBE_EXC provides support for reporting trace buffer management events as TRBE Profiling exceptions.

This feature is supported in AArch64 state only.

FEAT_TRBE_EXC is OPTIONAL from Armv9.5.

If FEAT_TRBE_EXC is implemented, then FEAT_TRBEv1p1 is implemented.

The following field identifies the presence of FEAT_TRBE_EXC:

- ID_AA64DFR2_EL1.TRBE_EXC.

FEAT_TRBEv1p1, Trace Buffer Extension version 1.1

FEAT_TRBEv1p1 introduces the following to the Trace Buffer Extension:

- If EL2 and FEAT_FGT are implemented, a fine-grained trap on the `TSB CSYNC` instruction.
- If EL2 is implemented, an EL2 control to override TRBLIMITR_EL1.nVM.
- Support for the TRBE Profiling exception extension, FEAT_TRBE_EXC.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in AArch64 state only.

FEAT_TRBEv1p1 is OPTIONAL from Armv9.5.

In an Armv9.6 implementation, if FEAT_TRBE is implemented, FEAT_TRBEv1p1 is implemented.

If FEAT_TRBEv1p1 is implemented, then FEAT_TRBE is implemented.

If FEAT_TRBEv1p1 is implemented, then FEAT_TRBE_EXC is implemented.

The following fields identify the presence of FEAT_TRBEv1p1:

- ID_AA64DFR0_EL1.TraceBuffer.
- TRBDEVARCH.REVISION.

FEAT_UINJ, Injection of Undefined Instruction exceptions

FEAT_UINJ introduces support for software injection of Undefined Instruction exceptions.

This feature is supported in AArch64 state only.

FEAT_UINJ is OPTIONAL from Armv9.0.

FEAT_UINJ is mandatory from Armv9.6.

The following field identifies the presence of FEAT_UINJ:

- ID_AA64PFR2_EL1.UINJ.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part B
The AArch64 Application Level Architecture

Chapter B1
The AArch64 Application Level Programmers' Model

This chapter describes the AArch64 application level programmers' model. It contains the following sections:

- About the Application level programmers' model.
- Registers in AArch64 Execution state.
- Process state, PSTATE.
- The Scalable Vector and Scalable Matrix Extensions (SVE & SME).
- Software control features and EL0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of the system information is needed to put the Application level programmers' model into context.

Depending on the implementation choices, the architecture supports multiple levels of execution privilege, indicated by different *Exception levels* that number upwards from EL0 to EL3. EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers' model is the programmers' model for software executing at EL0. For more information, see Exception levels.

System software determines the Exception level, and therefore the level of privilege, at which software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged at EL0. This:

- Permits the operating system to allocate system resources to an application in a unique or shared manner.
- Provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some system level understanding is necessary, and where relevant it gives a reference to the system level description.

Execution at any Exception level above EL0 is often referred to as privileged execution.

For more information on the system level view of the architecture refer to The AArch64 System Level Programmers' Model.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.2 Registers in AArch64 Execution state

The following registers are visible at EL0 using AArch64:

R0-R30 31 general-purpose registers, R0 to R30. Each can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

Figure B1-1 General-purpose register naming

The X30 general-purpose register is used as the procedure call link register.

SP A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack pointer can be accessed using the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

Note

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information, see the *Procedure Call Standard for the Arm 64-bit Architecture*.

PC A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can be updated only on a branch, exception entry or exception return.

Note

Attempting to execute an A64 instruction that is not word-aligned generates a PC alignment fault, see PC alignment checking.

V0-V31 32 SIMD&FP registers, V0 to V31. Each can be accessed as:

- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.
- A 128-bit vector of elements. See Figure A1-1.
- A 64-bit vector of elements. See Figure A1-1.

Where the number of bits described by a register name does not occupy an entire SIMD&FP register, it refers to the least significant bits. See Figure B1-2.

For more information about data types and vector formats, see Supported data types.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-2 SIMD and floating-point register naming

FPCR, FPSR The FPCR is the floating-point control register. The FPSR is the floating-point status register.

Z0-Z31 32 SVE scalable vector registers, Z0 to Z31, of equal length. Each register can be accessed as:

- A configurable-length vector of elements. The length, VL, is a power of two, from a minimum of 128 bits to an IMPLEMENTATION DEFINED maximum no greater than 2048 bits. See Figure B1-3, Figure A1-5, and Configurable SVE vector lengths.
- A SIMD&FP register, as described in V0-V31. Bits[127:0] of each Zn register hold the correspondingly numbered V0-V31 SIMD&FP register, as Figure B1-3 shows:

Figure B1-3 SVE scalable vector register naming

See also:

- Maximum implemented SVE vector lengths.
- Configurable SVE vector lengths.
- Treatment of SVE scalable vector registers.
- SVE writes of scalar values to registers.

P0-P15 16 SVE predicate registers, named P0 to P15. Each SVE predicate register holds one bit for each byte of an SVE scalar vector register.

Note

The *Maximum implemented SVE predicate length* is the *Maximum implemented SVE vector length* divided by 8. See Maximum implemented SVE vector lengths.

Also see Vector predication.

FFR The dedicated SVE First Fault Register that has the same size and format as the SVE predicate registers, P0-P15. See FFR, First Fault Register.

ZA Architectural state capable of holding a two-dimensional array of bytes. See ZA storage.

ZT0 A 512-bit SME2 lookup table register. See SME2 ZT0 register.

See also:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- System registers.
- Pseudocode description of registers in AArch64 state.
- Registers for instruction processing and exception handling.

B1.2.1 FFR, First Fault Register

R_{TRLWH} SVE has a dedicated First Fault Register named FFR.

I_{XPLQW} The FFR captures the cumulative fault status of a sequence of SVE First-fault and Non-fault vector load instructions.

R_{CPQQN} The FFR and the predicate registers have the same size and format.

I_{PBWPM} The FFR is a Special-purpose register.

R_{CGHCK} All bits in the FFR that are accessible at the current Exception level are initialized to 1 by using the `SETFFR` instruction.

R_{WZJVT} Bits in the FFR are indirectly set to 0 as a result of a suppressed access or suppressed fault corresponding to an *Active element* of an SVE First-fault or Non-fault vector load.

R_{BZLJG} Bits in the FFR are never set to 1 as a result of a vector load instruction.

I_{XLZQY} After a sequence of one or more SVE First-fault or Non-fault loads that follow a `SETFFR` instruction, without any intervening `WRFFR` instruction, the FFR contains a monotonic predicate value.

D_{SRDGX} A monotonic predicate value is one that consists of, starting from predicate element 0, a sequence of zero or more TRUE elements, followed only by zero or more FALSE elements.

I_{TQMTV} The TRUE elements in the FFR indicate the shortest sequence of consecutive elements that could contain valid data loaded from memory.

R_{GHFRQ} The only instructions that directly read the FFR are:

- `RDFFR`.
- `RDFFRS`.

R_{LHBRN} The only instructions that directly write the FFR are:

- `WRFFR`.
- `SETFFR`.

I_{DTHXN} The `WRFFR` instruction requires that its source predicate contains a monotonic predicate value. If the source is not a monotonic predicate value, then the resulting value in the FFR is UNKNOWN.

R_{XXMMP} All direct and indirect reads and writes to the FFR occur in program order relative to other instructions, without explicit synchronization.

B1.2.2 System registers

System registers provide support for execution control, status and general system configuration. The majority of the System registers are not accessible at EL0.

However, some System registers can be configured to allow access from software executing at EL0. Any access from EL0 to a System register without access permission causes the instruction to be trapped. The registers that can be accessed from EL0 are:

Debug registers A Debug Communications Channel is supported by the MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0, DLR_EL0 and DSPSR_EL0 registers.

General system control registers

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CTR_EL0 and DCZID_EL0 registers provide implementation parameters for EL0 cache management support.
- The SCXTNUM_EL0 register provides a number that can be used to separate out different context numbers.
- The POR_EL0 Permission Overlay register determines the unprivileged stage 1 overlay permissions for the EL1&0 and EL2&0 translation regimes.
- The TPIDR_EL0, TPIDR2_EL0, and TPIDRRO_EL0 registers provide locations where software can store thread identifying information.

Generic timer registers

- Read access to the system counter effective frequency using CNTFRQ_EL0.
- Physical and virtual timer count registers, CNTPCT_EL0, CNTVCT_EL0, CNTPCTSS_EL0 and CNTVCTSS_EL0.
- Physical up-count comparison, down-count value and timer control registers, CNTP_CVAL_EL0, CNTP_TVAL_EL0, and CNTP_CTL_EL0.
- Virtual up-count comparison, down-count value and timer control registers, CNTV_CVAL_EL0, CNTV_TVAL_EL0, and CNTV_CTL_EL0.

Performance Monitors registers

The Performance Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure some of these registers to be accessible at EL0.

For more details, see The Performance Monitors Extension.

Activity Monitors registers

The Activity Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure these registers to be accessible at EL0.

For more details, see The Activity Monitors Extension.

B1.2.3 Pseudocode description of registers in AArch64 state

The AArch64 pseudocode uses getter and setter accessor functions to read and write register files that are visible at EL0:

- The setter or assignment accessor, for example "`X[num, width] = result`", is used for register writes.
- The getter or non-assignment accessor, "`operand = X[num, width]`", is used for register reads.
- The *num* parameter specifies the register number to be accessed within the named register file, and *width* specifies the bit width of the register access.
- For the general-purpose register `X[]` accessor, register number 31 accesses the zero register, ZR, which reads as zero and ignores writes.
- For other register accessors, such as the SIMD&FP `V[]` accessors, there is no special interpretation of the register number.

Other special AArch64 accessors include:

- The `SP[]` accessors have no parameters and are used to read or write the currently selected 64-bit stack pointer at the current Exception level.
- The `PC64[]` accessor has no parameter list and is used to read the 64-bit Program Counter.
- The `Vpart[]` accessors are used to perform a partial read or write of an Advanced SIMD vector register, using an additional *part* parameter to select which half of the vector register to access.
- The `Z[]`, `V[]`, and `Vpart[]` accessors read and write the same underlying vector register file.

Other register accessors are defined as required by other architectural features. See A-profile Architecture Pseudocode, sections aarch64/functions/registers/.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.3 Process state, PSTATE

Process state, or PSTATE, is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

For the system level view of PSTATE, see Process state, PSTATE in The AArch64 System Level Programmers' Model.

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C and V Condition flags, combining them with the Condition code for the instruction to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see Condition flags and related instructions.

The exception masking bits

D Debug exception mask bit. When EL0 is enabled to modify the mask bits, this bit is visible and can be modified. However, this bit is architecturally ignored at EL0.

A SError mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

For each bit, the values are:

0 Exception not masked.

1 Exception masked.

Access at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

SME enable controls

SM *Streaming SVE mode* enable:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0 Streaming SVE mode not enabled.

1 Streaming SVE mode enabled.

See Streaming SVE mode.

ZA *ZA storage* and *ZT0 register* enable:

0 ZA storage and the ZT0 register are not enabled.

1 ZA storage and the ZT0 register are enabled.

See About PSTATE.ZA, ZA storage and SME2 ZT0 register.

B1.3.1 Accessing PSTATE fields at EL0

At EL0 using AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly read using the MRS instruction and directly written using the MSR (register) instructions. Table B1-1 shows the Special-purpose registers that access the PSTATE fields that hold AArch64 state when the PE is at EL0 using AArch64. All other PSTATE fields do not have direct read and write access at EL0.

Table B1-1 Accessing PSTATE fields at EL0 using MRS and MSR (register)

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
SVCR	SM, ZA
SSBS	SSBS
DIT	DIT
TCO	TCO

Software can also use the MSR (immediate) instruction to directly write to PSTATE.{D, A, I, F, SM, ZA, SSBS, DIT, TCO}. Table B1-2 shows the MSR (immediate) operands that can directly write to PSTATE.{D, A, I, F, SM, ZA, SSBS, DIT, TCO} when the PE is at EL0 using AArch64 state.

Table B1-2 Accessing PSTATE.{D, A, I, F, SM, ZA, SSBS, DIT, TCO} at EL0 using MSR (immediate)

Operand	PSTATE fields	Notes
DAIFSet	D, A, I, F	Directly sets any of the PSTATE.{D,A, I, F} bits to 1
DAIFClr	D, A, I, F	Directly clears any of the PSTATE.{D, A, I, F} bits to 0
SVCRSM	SM	Directly sets the PSTATE.SM bit to CRm<0>.
SVCRZA	ZA	Directly sets the PSTATE.ZA bit to CRm<0>.
SVCRSMZA	SM, ZA	Directly sets the PSTATE.{SM, ZA} bits to CRm<0>.
SSBS	SSBS	Directly sets the PSTATE.SSBS bit to CRm<0>
DIT	DIT	Directly sets the PSTATE.DIT bit to CRm<0>
TCO	TCO	Directly sets the PSTATE.TCO bit to CRm<0>

Access to the PSTATE.{D, A, I, F} fields at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Software can more efficiently change PSTATE.{SM, ZA} by using SMSTART and SMSTOP instructions.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects, are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

B1.3.2 SVE use of PSTATE N, Z, C, and V Condition flags

I_{WYXLS} This section describes the SVE-specific use of PSTATE.

I_{YZYCQ} PSTATE N, Z, C and V condition flags can be updated by any of the following:

- An SVE instruction that generates a predicate result and updates the PSTATE N, Z, C and V Condition flags based on the value of the result.
- An SVE instruction that updates the PSTATE N, Z, C and V Condition flags based on the value in its predicate source register or FFR:
  - PTEST.
  - RDFFRS (predicated).
- An SVE instruction that updates the PSTATE N, Z, C and V Condition flags based on the values in its general-purpose source registers:
  - CTERMEQ.
  - CTERMNE.

R_{TPXTF} When setting the PSTATE N, Z, C and V Condition flags for SVE predicated flag-setting instructions, the instruction's *Governing predicate* determines which predicate elements are considered Active.

R_{QJBRW} When setting the PSTATE N, Z, C and V Condition flags for SVE unpredicated flag-setting instructions, all predicate elements are considered Active.

R_{ZMRXC} Unless otherwise specified in an instruction description, the SVE flag-setting instructions update the PSTATE N, Z, C and V Condition flags as follows:

Flag	SVE Name	SVE interpretation
N	First	Set to 1 if the *First active element* was TRUE, otherwise cleared to 0.
Z	None	Cleared to 0 if any *Active element* was TRUE, otherwise set to 1.
C	Not last	Cleared to 0 if the *Last active element* was TRUE, otherwise set to 1.
V	-	Cleared to 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4 The Scalable Vector and Scalable Matrix Extensions (SVE & SME)

This section contains:

- Maximum implemented SVE vector lengths.
- Configurable SVE vector lengths.
- Treatment of SVE scalable vector registers.
- SVE writes of scalar values to registers.
- Vector predication.
- Streaming SVE mode.
- About PSTATE.ZA.
- ZA storage.
- ZA array vector access.
- ZA tile access.
- ZA storage layout.
- SME2 Multi-vector operands.
- SME2 ZT0 register.

B1.4.1 Maximum implemented SVE vector lengths

R_{QSVCQ} There are the following IMPLEMENTATION DEFINED vector lengths for the SVE scalable vector registers, Z0-Z31:

- A *Maximum implemented Streaming SVE vector length*.
- A *Maximum implemented Non-streaming SVE vector length*.

For each IMPLEMENTATION DEFINED vector length, all the following apply:

- The smallest architecturally defined maximum length is 128 bits.
- The largest architecturally defined maximum length is 2048 bits.
- The length is a power of two.

I_{GZRPL} There is no requirement for the Maximum implemented Streaming SVE vector length to be greater than or equal to the Maximum Non-streaming implemented SVE vector length.

I_{SGCVT} Where this architecture specification uses the term *Maximum implemented SVE vector length*, it means:

- The Maximum implemented Streaming SVE vector length when the PE is in *Streaming SVE mode*.
- The Maximum implemented Non-streaming SVE vector length otherwise.

See also:

- Z0-Z31.
- Configurable SVE vector lengths.

B1.4.2 Configurable SVE vector lengths

R_{KFTHG} There are the following configurable vector lengths for the SVE scalable vector registers, Z0-Z31:

- *Effective Streaming SVE vector length (SVL)*.
- *Effective Non-streaming SVE vector length (NSVL)*.

I_{QQRNR} SMCR_ELx.LEN requests an SVL. The architecturally defined SVL set is all powers of two from 128 to 2048 bits inclusive. Implementing any subset is permitted, up to and including the *Maximum implemented Streaming SVE vector length*. The subset is not required to be contiguous and is not required to start at 128 bits. Implementing a single SVL is permitted.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{NWYBP} ZCR_ELx.LEN requests an NSVL. The architecturally defined NSVL set is all powers of two from 128 to 2048 bits inclusive. An implementation is required to implement all architecturally defined values up to and including the *Maximum implemented Non-streaming SVE vector length*.

I_{WDKGR} An implementation is permitted to implement an SVL set that does not overlap with the implemented NSVL set.

R_{GWVHP} When SMCR_ELx.LEN requests an SVL, the PE selects an SVL according to the steps described in the SMCR_ELx.LEN field descriptions.

R_{MMCTJ} When ZCR_ELx.LEN requests an NSVL, the PE selects an NSVL according to the steps described in the ZCR_ELx.LEN field descriptions.

I_{VCQBB} Where this architecture specification uses the term *Effective SVE vector length (VL)*, it means:

- SVL when the PE is in *Streaming SVE mode*.
- NSVL otherwise.

R_{PXZTM} When executing at an Exception level where VL is less than the *Maximum implemented SVE vector length*, bits VL and higher in the SVE scalable vector registers, and bits VL ÷ 8 and higher in the SVE predicate registers and FFR, are inaccessible.

R_{YRPDH} When VL is increased, it is CONSTRAINED UNPREDICTABLE whether the previously inaccessible bits that become accessible have:

- A value of zero.
- The value they had before executing at the more constrained length.

I_{HGYPV} VL might increase because of an explicit action such as writing to SMCR_ELx.LEN or ZCR_ELx.LEN, or an implicit action such as taking an exception to an Exception level with a less constrained VL.

I_{PDLWX} When changing VL, the contents in the bits that consistently remain accessible remain the same.

R_{KXKNK} VL at Exception level ELx is 128 bits if all the following are true:

- SVE instructions are disabled or trapped at ELx, or are not available because ELx is using AArch32.
- Floating-point instructions are not trapped at ELx.

R_{DMBPN} When taking an exception to a target Exception level, if SVE, SME, and floating-point instructions are disabled, trapped, or not available at all Exception levels below the target Exception level, for the current Security state, the SVE register state at the target Exception level is preserved.

I_{FMWZZ} When VL is increased, steps must be taken to ensure the newly accessible area does not expose values unrelated to another body of software in a different trust or security scope. This might be achieved by, for example, ensuring that the previously inaccessible bits in the SVE Z, P, and FFR registers, and the SME ZA storage and ZT0 register, are reset to zero before they are next used.

I_{BRMMV} System software provides a maximum VL to lower-privileged software, which might further constrain the VL. However, system software must initialize and context switch values consistent with the maximum VL it provides and should not make assumptions about any smaller length that lower-privileged software is using. For example, if a hypervisor exposes a VL of 512 to a VM, that VM might choose to constrain VL to 256. The hypervisor must still save and restore 512-bit vectors to prevent leakage of values between VMs, because the VM might later raise its SVL to 512 and must not be able to observe values created by other software in the newly visible upper portion of the registers.

See also:

- Maximum implemented SVE vector lengths.
- Streaming SVE mode.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4.3 Treatment of SVE scalable vector registers

I_{GKWYJ} Unless stated otherwise in an instruction description, SVE instructions treat an SVE scalable vector register as containing one or more *vector elements* of equal size.

I_{CDKJQ} SVE instructions can process vector elements in parallel, unless an instruction description states otherwise.

R_{CJZLM} If the order of operations performed by an SVE instruction on vector or predicate elements has observable significance, elements are processed in increasing element number order.

R_{KHDBN} When an SVE instruction treats an SVE scalable vector register as containing multiple vector elements, the element size is encoded in the opcode of the instruction. The element size is 8, 16, 32, 64, or 128 bits.

R_{WKYLB} When the *Effective SVE vector length* (VL) at the current Exception level is greater than 128 bits, any AArch64 instruction that writes to an 128-bit SIMD&FP register, V0-V31, sets bits [VL-1:128] to zero.

B1.4.4 SVE writes of scalar values to registers

I_{ZDLGD} Certain SVE instructions generate a scalar result that is written to an AArch64 general-purpose register or to element[0] of a vector register.

R_{HNVTM} When an SVE instruction generates a scalar result of width N bits, the instruction places the result in bits [N-1:0] of the destination register.

R_{QCLSH} When an SVE instruction generates a scalar result of width N bits, and N is less than the maximum accessible destination register width RW, the instruction sets bits [RW-1:N] of the destination register to zero.

B1.4.5 Vector predication

R_{JFJZX} There are the following vector predication concepts:

- Predicate-as-mask, if FEAT_SVE or FEAT_SME is implemented.
- Predicate-as-counter, if FEAT_SME2 or FEAT_SVE2p1 is implemented.

I_{HWRFM} The assembler syntax for a predicate register is of the form:

- *Pg* for predicate-as-mask.
- *PNg* for predicate-as-counter.

Pg and PNg refer to the same SVE predicate register.

The remainder of this section contains:

- Predicate-as-mask.
- Predicate-as-counter.
- SVE predicated instructions.

B1.4.5.1 Predicate-as-mask

I_{JHCTW} Most SVE instructions interpret SVE predicate registers as a predicate-as-mask encoding.

R_{XMPLM} A predicate-as-mask encoding is divided into 1-bit, 2-bit, 4-bit, or 8-bit *predicate elements*. Each predicate element corresponds to a vector element in an SVE scalable vector register.

R_{XVRKX} Unless stated otherwise in an instruction description, SVE instructions treat an SVE predicate register as containing one or more predicate elements of equal size.

R_{XCZQR} When an SVE instruction treats an SVE predicate register as containing multiple predicate elements, the element size is encoded in the opcode of the instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{HRPMD} If the lowest-numbered bit of a predicate element is 0, the value of the predicate element is FALSE.

R_{DNMFH} If the lowest-numbered bit of a predicate element is 1, the value of the predicate element is TRUE.

R_{LTGQC} For SVE instructions that generate predicate-as-mask encodings, if all the following are true, all bits except the lowest-numbered bit of each destination predicate element are set to zero:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

R_{HBMLS} For SVE instructions that consume predicate-as-mask encodings, if all the following are true, all bits except the lowest-numbered bit of each source predicate element are ignored on reads:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

B1.4.5.2 Predicate-as-counter

R_{DSFKR} SME2 and SVE2.1 multi-vector instructions interpret bits[15:0] of SVE predicate registers as a *predicate-as-counter encoding*:

Table B1-4 Predicate-as-counter encoding

Bit[15] Invert	Bits[14:(LSZ+1)][a] Element count[b]	Bits[LSZ:0] Element size	Meaning
`0bX`	`0bXXXXXXXXXXXXXXX`	`0b1`	Byte-size elements
`0bX`	`0bXXXXXXXXXXXXXX`	`0b10`	Halfword-size elements
`0bX`	`0bXXXXXXXXXXXXX`	`0b100`	Word-size elements
`0bX`	`0bXXXXXXXXXXXX`	`0b1000`	Doubleword-size elements
`0bX`	`0bXXXXXXXXXXXX`	`0b0000`	Any-size elements. All FALSE predicate.

a. LSZ = the number of trailing zeroes in the element size field.
b. The 4 most significant bits are reserved.

Element size, bits[LSZ:0]

For values shown in Table B1-4 other than `0b0000`:

- Number of trailing zeroes = $\log_2$(element size in bytes).
  For example, 2 trailing zeroes = $\log_2$(4 bytes). This means word-size elements.

Element count, bits[14:(LSZ+1)]

An unsigned integer value:

- The number of consecutive elements, starting from element 0, that are TRUE or FALSE according to bit[15].

Invert, bit[15]

`0b0` [14:(LSZ+1)] is holding the number of TRUE elements. All other elements are FALSE.

`0b1` [14:(LSZ+1)] is holding the number of FALSE elements. All other elements are TRUE.

I_{ZPLKP} The maximum number of elements that the predicate-as-counter encoding can count is four vectors of byte elements, where the length of each vector is the largest architecturally defined length, 2048 bits:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table B1-5

Bit[15] Invert	Bits[14:(LSZ+1)] Element count	Bits[LSZ:0] Element size	Total number of bits
1 bit	4 reserved bits. 10 count bits. 10 count bits can count 1024 elements, which is the number of byte elements in four 2048-bit long vectors.	1 bit	16 bits

D_{BJMYH} The canonical all-FALSE predicate-as-counter encoding has an element count of zero, with the invert bit set to 0 and an element size field set to `0b0000`.

D_{JGSYR} The canonical all-TRUE predicate-as-counter encoding has an element count of zero, with the invert bit set to 1 and a nonzero element size field determined by the generating instruction.

R_{YCRMW} If the *Effective SVE vector length* (VL) is greater than 128 bits, then an instruction generating a predicate-as-counter encoding sets bits 16 and higher to zero.

R_{SGVTC} If VL is greater than 128 bits, then an instruction consuming a predicate-as-counter encoding ignores bits 16 and higher.

R_{YSTVC} An instruction consuming a predicate-as-counter encoding uses the least significant bits of the element count field required to count the number of bytes in VL times four. The instruction ignores the more significant bits in the element count field. Table B1-6 gives an example for VL = 512 bits and byte-size elements:

Table B1-6

Bit[15]Invert	Bits[14:(LSZ+1)] Element count	Bits[LSZ:0] Element size
1 bit	4 reserved bits, bits[14:11]. The instruction ignores these. 2 bits, bits[10:9]. The instruction ignores these. 8 count bits, bits[8:1]. The instruction uses these. 8 count bits can count 256 elements, which is the number of byte elements in four 512-bit long vectors.	1 bit

See also:

- P0-P15.
- Configurable SVE vector lengths.

B1.4.5.3 SVE predicated instructions

I_{VKHDR} If an instruction supports predication, it is known as a *predicated instruction*.

I_{FNFRN} The predicate operand that is used to determine the *Active elements* of a predicated instruction is known as the *Governing predicate*.

I_{SVYXB} An instruction that does not have a Governing predicate operand and implicitly treats all other vector and predicate elements as Active is known as an *unpredicated instruction*.

I_{KNKBN} Many predicated instructions can only use P0-P7 as the Governing predicate.

R_{LZVFJ} When a Governing predicate element is TRUE, the corresponding element in other vector or predicate operands is an Active element.

R_{CNFLG} When a Governing predicate element is FALSE, the corresponding element in other vector or predicate operands is an *Inactive element*.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{CBYJH} Predicated instructions process Active elements.

R_{LDXSF} Predicated instructions do not process Inactive elements.

R_{GJLPZ} Unpredicated instructions process all elements in their vector or predicate operands.

R_{WLQBD} When a predicated instruction writes to a vector destination register or a predicate destination register, one of the following happens:

- The Inactive elements in the destination register are set to zero.
- The Inactive elements in the destination register retain their previous value.

I_{QBHRN} Zeroing predication is performed when the Inactive elements in the destination register are set to zero.

I_{YPYRF} Merging predication is performed when Inactive elements in the destination register retain their previous value.

B1.4.6 Streaming SVE mode

R_{XDPXS} A PE with FEAT_SME has *Streaming SVE mode*.

I_{ZTTNW} In Streaming SVE mode:

- SME and SME2 instructions access *Streaming SVE register state*, which comprises:
  - SVE scalable vector registers, Z0 to Z31.
  - SVE predicate registers, P0 to P15.
  - SVE First Fault register, FFR, if FEAT_SME_FA64 is implemented and enabled at the current Exception level.
- PSTATE.ZA can be toggled, to enable and disable the *ZA storage* and *ZT0 register* as required.

R_{RSWFQ} When the Effective value of PSTATE.SM is changed by any method from 0 to 1, an entry to Streaming SVE mode is performed, and all implemented bits of Streaming SVE register state are set to zero.

R_{KFRQZ} When the Effective value of PSTATE.SM is changed by any method from 1 to 0, an exit from Streaming SVE mode is performed, and in the newly-entered mode, all implemented bits of the SVE scalable vector registers, SVE predicate registers, and FFR, are set to zero.

I_{NHNFF} A *legal* instruction is an instruction that the architecture permits execution of when PSTATE.{SM, ZA} are in the required state, unless its execution at the current Exception level is prevented by a configurable trap or enable.

I_{HZFSG} An *illegal* instruction is an instruction whose attempted execution by a PE when PSTATE.SM and PSTATE.ZA are not in the required state causes an *SME illegal instruction exception* to be taken, unless its execution at the current Exception level is prevented by a higher-priority configurable trap or enable.

R_{VJZBC} In Streaming SVE mode:

- Streaming SVE register state is valid, and SME and SME2 instructions that access it are legal.
- When PSTATE.ZA is 1, SME and SME2 instructions that access ZA storage or the ZT0 register are legal.
- SME and SME2 instructions that do not access ZA storage or the ZT0 register are legal.
- Legal instructions that access the SVE scalar vector registers or their correspondingly numbered SIMD&FP registers access Streaming SVE register state.
- The following are illegal if FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:
  - Some SVE and SVE2 instructions. The individual instruction descriptions in SVE Instruction Descriptions describe which are illegal.
  - All those instructions that call the AArch64.CheckFPAdvSIMDEnabled() function.

R_{XBBFD} When the PE is not in Streaming SVE mode:

- Streaming SVE register state is not valid, and SME and SME2 instructions that access it are illegal.
- When PSTATE.ZA is 1:
  - SME `LDR` (array vector), `STR` (array vector), and `ZERO` (tiles) instructions that access ZA storage are legal. All other instructions that access ZA storage are illegal.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— SME2 `LDR` (table), `STR` (table), and `ZERO` (table) instructions that access the ZT0 register are legal. All other instructions that access the ZT0 register are illegal.

I_{TDSPN} In Streaming SVE mode, the following instructions might be significantly delayed if FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:

- Instructions which are dependent on results generated from vector or SIMD&FP register sources written to a general-purpose destination register, a predicate destination register, or the NZCV condition flags.

The Operational information sections of instruction descriptions in A64 Advanced SIMD and Floating-point Instruction Descriptions and SVE Instruction Descriptions describe which instructions are affected.

I_{MHTLZ} When the Effective value of PSTATE.SM is changed by any method from 0 to 1 or from 1 to 0, the FPSR is set to the value `0x0000_0000_0800_009f`, setting all the cumulative status bits to 1.

I_{YTZVD} Statements which refer to the value of the SVE scalable vector registers, Z0-Z31, implicitly also refer to the lower bits of those registers accessed by the SIMD&FP register names V0-V31, Q0-Q31, D0-D31, S0-S31, H0-H31, and B0-B31.

See also:

- Registers in AArch64 Execution state.
- About PSTATE.ZA.
- Accessing PSTATE fields at EL0.
- Floating-point behaviors in Streaming SVE mode.

B1.4.7 About PSTATE.ZA

I_{HVKTL} PSTATE.ZA enables ZA storage and, if FEAT_SME2 is implemented, the ZT0 register.

R_{JHMYL} When PSTATE.ZA is 0, the contents of ZA storage and the ZT0 register are not valid, and SME and SME2 instructions that access them are *illegal*.

R_{SFWMY} When PSTATE.ZA is 1, the contents of ZA storage and the ZT0 register are valid and are retained by hardware irrespective of whether the PE is in *Streaming SVE mode*, and:

- SME `LDR` (array vector), `STR` (array vector), and `ZERO` (tiles) instructions that access ZA storage are legal.
- SME2 `LDR` (table), `STR` (table), and `ZERO` (table) instructions that access the ZT0 register are legal.
- When the PE is in Streaming SVE mode, all other SME and SME2 instructions that access ZA storage or the ZT0 register are legal.

R_{YRZRM} When PSTATE.ZA is changed by any method from 0 to 1, all implemented bits of ZA storage and the ZT0 register are set to zero.

I_{LRDZR} When PSTATE.ZA is changed from 1 to 0, there is no architecturally defined effect on ZA storage and the ZT0 register, because the contents of ZA storage and the ZT0 register cannot be observed when PSTATE.ZA is 0.

I_{QWCJS} When PSTATE.ZA is changed from 0 to 1, or from 1 to 0, there is no effect on the SVE scalable vector registers, SVE predicate registers, and FPSR if PSTATE.SM is not changed.

See also:

- Streaming SVE mode.
- ZA storage.
- SME2 ZT0 register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4.8 ZA storage

R_{FFWLL} A PE with FEAT_SME has *ZA storage*.

D_{JBVYJ} There are the following terms for the number of elements in a vector of SVL bits:

SVL_B The number of byte elements, SVL ÷ 8.

SVL_H The number of halfword elements, SVL ÷ 16.

SVL_S The number of word elements, SVL ÷ 32.

SVL_D The number of doubleword elements, SVL ÷ 64.

SVL_Q The number of quadword elements, SVL ÷ 128.

R_{SSXPL} ZA storage is architectural register state consisting of a two-dimensional ZA array of [$SVL_B \times SVL_B$] bytes.

See also:

- About PSTATE.ZA.
- Accessing PSTATE fields at EL0.
- Streaming SVE mode.

B1.4.9 ZA array vector access

R_{FFWNB} The *ZA* array can be accessed as vectors of SVL bits.

D_{PPPCM} An untyped vector access to the *ZA* array is represented by ZA[*N*], where *N* is in the range 0 to SVL_B-1 inclusive.

D_{DTVZN} In SME `LDR` (array vector) and `STR` (array vector) instructions, an untyped *ZA array vector* is selected by the sum of a 32-bit general-purpose vector select register W*v* and an immediate vector select offset *offs*, modulo SVL_B.

D_{YXHFR} The preferred disassembly for an untyped *ZA* array vector is ZA[W*v*, *offs*], where *offs* is an immediate in the range 0-15 inclusive.

D_{CRJPC} The *ZA* array can be accessed as vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements.

D_{WMVZT} An elementwise vector access to the *ZA* array is indicated by appending a vector index "[*N*]" to the *ZA* array name and element size qualifier, where *N* is in the range 0 to SVL_B-1 inclusive, as follows:

- An 8-bit element vector access to the *ZA* array is represented by ZA.B[*N*].
- A 16-bit element vector access to the *ZA* array is represented by ZA.H[*N*].
- A 32-bit element vector access to the *ZA* array is represented by ZA.S[*N*].
- A 64-bit element vector access to the *ZA* array is represented by ZA.D[*N*].
- A 128-bit element vector access to the *ZA* array is represented by ZA.Q[*N*].

B1.4.10 ZA tile access

D_{VSVMX} A *ZA tile* is a square, two-dimensional sub-array of elements within the *ZA* array.

I_{WLRTV} Depending on the element size with which it is accessed, the ZA array is treated as containing one or more *ZA* tiles, as described in the following sections.

D_{DWMYT} A *ZA* tile is indicated by appending the tile number to the *ZA* name.

D_{ZGBHT} A *ZA tile slice* is a one-dimensional set of horizontally or vertically contiguous elements within a *ZA* tile.

R_{PZNWB} A vector access to a tile reads or writes a *ZA* tile slice.

I_{NFXHH} A *ZA* tile can be accessed as vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{YZDBS} A *ZA* tile can be accessed as horizontal slices of SVL bits.

R_{GPVSZ} A *ZA* tile is accessed as horizontal slices if the V field in the accessing instruction opcode is 0.

D_{TRHTX} An access to horizontal tile slices is indicated by an "H" suffix on the *ZA* tile name.

I_{HBYTT} A *ZA* tile can be accessed as vertical slices of SVL bits.

R_{GPPPK} A *ZA* tile is accessed as vertical slices if the V field in the accessing instruction opcode is 1.

D_{WSBVG} An access to vertical tile slices is indicated by a suffix on the *ZA* tile name.

R_{TWWTL} In SME instructions, the tile slice is selected by the sum of a 32-bit general-purpose slice index register W*s* and an immediate slice index offset *offs*, modulo the number of slices in the named tile.

B1.4.10.1 Accessing an 8-bit element ZA tile

D_{HMSNH} An 8-bit element *ZA* tile is indicated by a ".B" qualifier following the tile name.

D_{NLCNH} There is a single tile named ZA0.B which consists of $[SVL_B \times SVL_B]$ 8-bit elements and occupies all of the *ZA* storage.

R_{NBSMJ} An access to a horizontal or vertical 8-bit element ZA tile slice reads or writes SVL_B 8-bit elements.

D_{NMHLM} An access to a horizontal or vertical 8-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *N* is in the range 0 to SVL_B-1 inclusive:

- ZA0H.B[*N*] indicates a horizontal 8-bit element *ZA* tile slice selection.
- ZA0V.B[*N*] indicates a vertical 8-bit element *ZA* tile slice selection.

I_{JVTNY} Horizontal and vertical ZA0.B slice accesses are illustrated in the following diagram for SVL of 256 bits:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-4 Horizontal and vertical ZA0.B slice for SVL of 256

R_{DCSDX} An access to the horizontal slice ZA0H.B[*N*] reads or writes the SVL_B bytes in *ZA* array vector ZA.B[*N*].

R_{FHYSQ} An access to the vertical slice ZA0V.B[*N*] reads or writes the 8-bit element [*N*] within each horizontal slice of ZA0.B.

D_{CDDVV} The preferred disassembly is:

- ZA0H.B[W*s*, *offs*], for a horizontal 8-bit element *ZA* tile slice selection.
- ZA0V.B[W*s*, *offs*], for a vertical 8-bit element *ZA* tile slice selection.

Where *offs* is an immediate in the range 0-15 inclusive.

B1.4.10.2 Accessing a 16-bit element ZA tile

D_{LNXPD} A 16-bit element *ZA* tile is indicated by a ".H" qualifier following the tile name.

D_{GWZDM} There are two tiles named ZA0.H and ZA1.H. Each tile consists of [$SVL_H \times SVL_H$] 16-bit elements, and occupies half of the *ZA* storage.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{NMGXG} An access to a horizontal or vertical 16-bit element ZA tile slice reads or writes SVL_H 16-bit elements.

D_{DHKMC} An access to a horizontal or vertical 16-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is 0 or 1, and *N* is in the range 0 to SVL_H-1 inclusive:

- ZA*t*H.H[*N*] indicates a horizontal 16-bit element *ZA* tile slice selection.
- ZA*t*V.H[*N*] indicates a vertical 16-bit element *ZA* tile slice selection.

I_{ZSWJW} Horizontal and vertical ZA*t*.H slice accesses, where *t* is 0 or 1, are illustrated in the following diagram for SVL of 256 bits:

Figure B1-5 Horizontal and vertical ZAt.H slice for SVL of 256

R_{BTLQC} An access to the horizontal slice ZA*t*H.H[*N*] reads or writes the SVL_H 16-bit elements in *ZA* array vector ZA.H[*t* + 2 * *N*].

R_{NGJBJ} An access to the vertical slice ZA*t*V.H[*N*] reads or writes the 16-bit element [*N*] within each horizontal slice of ZA*t*.H.

D_{RHQJT} The preferred disassembly is as follows:

- ZA*t*H.H[W*s*, *offs*], for a horizontal 16-bit element *ZA* tile slice selection.
- ZA*t*V.H[W*s*, *offs*], for a vertical 16-bit element *ZA* tile slice selection.

Where *t* is 0 or 1, and *offs* is an immediate in the range 0-7 inclusive.

B1.4.10.3 Accessing a 32-bit element ZA tile

D_{HBKZV} A 32-bit element *ZA* tile is indicated by a ".S" qualifier following the tile name.

D_{RDRRT} There are four tiles named ZA0.S, ZA1.S, ZA2.S, and ZA3.S. Each tile consists of [$SVL_S \times SVL_S$] 32-bit elements, and occupies a quarter of the *ZA* storage.

R_{XFPPL} An access to a horizontal or vertical 32-bit element ZA tile slice reads or writes SVL_S 32-bit elements.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D_{JFPSJ} An access to a horizontal or vertical 32-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is 0, 1, 2, or 3, and *N* is in the range 0 to SVL_S-1 inclusive:

- ZA*t*H.S[*N*] indicates a horizontal 32-bit element *ZA* tile slice selection.
- ZA*t*V.S[*N*] indicates a vertical 32-bit element *ZA* tile slice selection.

I_{SZXZR} Horizontal and vertical ZA*t*.S slice accesses, where *t* is 0, 1, 2, or 3, are illustrated in the following diagram for SVL of 256 bits:

Figure B1-6 Horizontal and vertical ZAt.S slice for SVL of 256

R_{JBJZY} An access to the horizontal slice ZA*t*H.S[*N*] reads or writes the SVL_S 32-bit elements in *ZA* array vector ZA.S[*t* + 4 * *N*].

R_{GBYSJ} An access to the vertical slice ZA*t*V.S[*N*] reads or writes the 32-bit element [*N*] within each horizontal slice of ZA*t*.S.

D_{LQLJH} The preferred disassembly is:

- ZA*t*H.S[W*s*, *offs*], for a horizontal 32-bit element *ZA* tile slice selection.
- ZA*t*V.S[W*s*, *offs*], for a vertical 32-bit element *ZA* tile slice selection.

Where *t* is 0, 1, 2, or 3, and *offs* is 0, 1, 2, or 3.

B1.4.10.4 Accessing a 64-bit element ZA tile

D_{TWMMM} A 64-bit element *ZA* tile is indicated by a ".D" qualifier following the tile name.

D_{THPSD} There are eight tiles named ZA0.D, ZA1.D, ZA2.D, ZA3.D, ZA4.D, ZA5.D, ZA6.D, and ZA7.D. Each tile consists of [$SVL_D \times SVL_D$] 64-bit elements, and occupies an eighth of the *ZA* storage.

R_{ZXYBQ} An access to a horizontal or vertical 64-bit element ZA tile slice reads or writes SVL_D 64-bit elements.

D_{DCXSX} An access to a horizontal or vertical 64-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is in the range 0-7 inclusive, and *N* is in the range 0 to SVL_D-1 inclusive:

- ZA*t*H.D[*N*] indicates a horizontal 64-bit element *ZA* tile slice selection.
- ZA*t*V.D[*N*] indicates a vertical 64-bit element *ZA* tile slice selection.

I_{LGJZC} Horizontal and vertical ZA*t*.D slice accesses, where *t* is in the range 0-7 inclusive, are illustrated in the following diagram for SVL of 256 bits:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-7 Horizontal and vertical ZAt.D slice for SVL of 256

R_{CVVJK} An access to the horizontal slice ZA*t*H.D[*N*] reads or writes the SVL_D 64-bit elements in *ZA* array vector ZA.D[*t* + 8 * *N*].

R_{JYQKK} An access to the vertical slice ZA*t*V.D[*N*] reads or writes the 64-bit element [*N*] within each horizontal slice of ZA*t*.D.

D_{MQQPX} The preferred disassembly is:

- ZA*t*H.D[W*s*, *offs*], for a horizontal 64-bit element *ZA* tile slice selection.
- ZA*t*V.D[W*s*, *offs*], for a vertical 64-bit element *ZA* tile slice selection.

Where *t* is in the range 0-7 inclusive, and *offs* is 0 or 1.

B1.4.10.5 Accessing a 128-bit element ZA tile

D_{GZDSH} A 128-bit element *ZA* tile is indicated by a ".Q" qualifier following the tile name.

D_{RPMJL} There are 16 tiles named ZA0.Q, ZA1.Q, ZA2.Q, ZA3.Q, ZA4.Q, ZA5.Q, ZA6.Q, ZA7.Q, ZA8.Q, ZA9.Q, ZA10.Q, ZA11.Q, ZA12.Q, ZA13.Q, ZA14.Q, and ZA15.Q. Each tile consists of [$SVL_Q \times SVL_Q$] 128-bit elements, and occupies 1/16 of the *ZA* storage.

R_{QGHPF} An access to a horizontal or vertical 128-bit element ZA tile slice reads or writes SVL_Q 128-bit elements.

D_{RLQKW} An access to a horizontal or vertical 128-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is in the range 0-15 inclusive, and *N* is in the range 0 to SVL_Q-1 inclusive:

- ZA*t*H.Q[*N*] indicates a horizontal 128-bit element *ZA* tile slice selection.
- ZA*t*V.Q[*N*] indicates a vertical 128-bit element *ZA* tile slice selection.

I_{YQPWS} Horizontal and vertical ZA*t*.Q slice accesses, where *t* is in the range 0-15 inclusive, are illustrated in the following diagram for SVL of 256 bits:

Figure B1-8 Horizontal and vertical ZAt.Q slice for SVL of 256

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{PJTQJ} An access to the horizontal slice ZA*t*H.Q[*N*] reads or writes the SVL_Q 128-bit elements in *ZA* array vector ZA.Q[*t* + 16 * *N*].

R_{TRJFZ} An access to the vertical slice ZA*t*V.Q[*N*] reads or writes the 128-bit element [*N*] within each horizontal slice of ZA*t*.Q.

D_{VCLJP} The preferred disassembly is:

- ZA*t*H.Q[W*s*, 0], for a horizontal 128-bit element *ZA* tile slice selection.
- ZA*t*V.Q[W*s*, 0], for a vertical 128-bit element *ZA* tile slice selection.

Where *t* is in the range 0-15 inclusive, and the slice index offset is always zero.

B1.4.11 ZA storage layout

B1.4.11.1 ZA array vector and tile slice mappings

I_{PYTLW} Each horizontal tile slice corresponds to one *ZA* array vector.
The horizontal slice mappings for all tile sizes are illustrated by this table:

ZA Array Vector	8-bit element Tile Horizontal Slice	16-bit element Tile Horizontal Slice	32-bit element Tile Horizontal Slice	64-bit element Tile Horizontal Slice	128-bit element Tile Horizontal Slice
ZA[0]	ZA0H.B[0]	ZA0H.H[0]	ZA0H.S[0]	ZA0H.D[0]	ZA0H.Q[0]
ZA[1]	ZA0H.B[1]	ZA1H.H[0]	ZA1H.S[0]	ZA1H.D[0]	ZA1H.Q[0]
ZA[2]	ZA0H.B[2]	ZA0H.H[1]	ZA2H.S[0]	ZA2H.D[0]	ZA2H.Q[0]
ZA[3]	ZA0H.B[3]	ZA1H.H[1]	ZA3H.S[0]	ZA3H.D[0]	ZA3H.Q[0]
ZA[4]	ZA0H.B[4]	ZA0H.H[2]	ZA0H.S[1]	ZA4H.D[0]	ZA4H.Q[0]
ZA[5]	ZA0H.B[5]	ZA1H.H[2]	ZA1H.S[1]	ZA5H.D[0]	ZA5H.Q[0]
ZA[6]	ZA0H.B[6]	ZA0H.H[3]	ZA2H.S[1]	ZA6H.D[0]	ZA6H.Q[0]
ZA[7]	ZA0H.B[7]	ZA1H.H[3]	ZA3H.S[1]	ZA7H.D[0]	ZA7H.Q[0]
ZA[8]	ZA0H.B[8]	ZA0H.H[4]	ZA0H.S[2]	ZA0H.D[1]	ZA8H.Q[0]
ZA[9]	ZA0H.B[9]	ZA1H.H[4]	ZA1H.S[2]	ZA1H.D[1]	ZA9H.Q[0]
ZA[10]	ZA0H.B[10]	ZA0H.H[5]	ZA2H.S[2]	ZA2H.D[1]	ZA10H.Q[0]
ZA[11]	ZA0H.B[11]	ZA1H.H[5]	ZA3H.S[2]	ZA3H.D[1]	ZA11H.Q[0]
ZA[12]	ZA0H.B[12]	ZA0H.H[6]	ZA0H.S[3]	ZA4H.D[1]	ZA12H.Q[0]
ZA[13]	ZA0H.B[13]	ZA1H.H[6]	ZA1H.S[3]	ZA5H.D[1]	ZA13H.Q[0]
ZA[14]	ZA0H.B[14]	ZA0H.H[7]	ZA2H.S[3]	ZA6H.D[1]	ZA14H.Q[0]
ZA[15]	ZA0H.B[15]	ZA1H.H[7]	ZA3H.S[3]	ZA7H.D[1]	ZA15H.Q[0]
if applicable ZA[16] to ZA[SVL_B-1]	. . .	. . .	. . .	. . .	. . .

B1.4.11.2 Tile mappings

I_{YVYJP} The smallest *ZA* tile granule is the 128-bit element tile. When the *ZA* storage is viewed as an array of tiles, the larger 64-bit, 32-bit, 16-bit, and 8-bit element tiles overlap multiple 128-bit element tiles as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Tile	Overlaps
ZA0.B	ZA0.Q, ZA1.Q, ZA2.Q, ZA3.Q, ZA4.Q, ZA5.Q, ZA6.Q, ZA7.Q, ZA8.Q, ZA9.Q, ZA10.Q, ZA11.Q, ZA12.Q, ZA13.Q, ZA14.Q, ZA15.Q
ZA0.H	ZA0.Q, ZA2.Q, ZA4.Q, ZA6.Q, ZA8.Q, ZA10.Q, ZA12.Q, ZA14.Q
ZA1.H	ZA1.Q, ZA3.Q, ZA5.Q, ZA7.Q, ZA9.Q, ZA11.Q, ZA13.Q, ZA15.Q
ZA0.S	ZA0.Q, ZA4.Q, ZA8.Q, ZA12.Q
ZA1.S	ZA1.Q, ZA5.Q, ZA9.Q, ZA13.Q
ZA2.S	ZA2.Q, ZA6.Q, ZA10.Q, ZA14.Q
ZA3.S	ZA3.Q, ZA7.Q, ZA11.Q, ZA15.Q
ZA0.D	ZA0.Q, ZA8.Q
ZA1.D	ZA1.Q, ZA9.Q
ZA2.D	ZA2.Q, ZA10.Q
ZA3.D	ZA3.Q, ZA11.Q
ZA4.D	ZA4.Q, ZA12.Q
ZA5.D	ZA5.Q, ZA13.Q
ZA6.D	ZA6.Q, ZA14.Q
ZA7.D	ZA7.Q, ZA15.Q

I_{WGZBT} The architecture permits concurrent use of different element size tiles.

B1.4.11.3 Horizontal tile slice mappings

I_{NJJXW} The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for a 32-bit element and 64-bit element horizontal tile slice.

Each small numbered square represents 8 bits.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-9 ZA storage mapping for SVL of 256 bits: 32-bit and 64-bit element horizontal tile slice

An SME vector load, store, or move instruction that accesses horizontal tile slices ZA2H.S[1] or ZA4H.D[2] treats the slices as vectors with the following layout:

Figure B1-10 ZA2H.S[1] and ZA4H.D[2] vector layout

B1.4.11.4 Vertical tile slice mappings

I_{TNCCV}

The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for a 32-bit element and 64-bit element vertical tile slice.

Each small numbered square represents 8 bits.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-11 ZA storage mapping for SVL of 256 bits: 32-bit and 64-bit element vertical tile slices

An SME vector load, store, or move instruction which accesses vertical tile slices ZA2V.S[1] or ZA4V.D[2] treats the slices as vectors with the following layout:

Figure B1-12 ZA2V.S[1] and ZA4V.D[2] vector layout

B1.4.11.5 Mixed horizontal and vertical tile slice mappings

ICGXPJ

The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for various element size tiles, horizontal tile slices, and vertical tile slices.

Each small square represents 8 bits.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-13 ZA storage mapping for SVL of 256 bits: various vertical and tile slices

I_{HVFMB} It is possible to simultaneously use non-overlapping *ZA* array vectors within tiles of differing element sizes. For example, tiles ZA1.H, ZA0.S, and ZA2.D have no *ZA* array vectors in common, as illustrated in the following diagram for SVL of 256 bits:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-14 Using various non-overlapping ZA array vectors within tiles of different element sizes

IWDMCK It is possible to access overlapping *ZA* array vectors within tiles of differing element sizes. For example, tiles ZA0.H, ZA2.S, and ZA6.D have common *ZA* array vectors.

B1.4.12 SME2 Multi-vector operands

RKLRKJ Multi-vector operands allow certain SME2 instructions to access as source and destination operands:

- A group of two or four SVE *Z* vector registers.
- A group of two or four *ZA* tile slices.
- A group of two, four, eight, or sixteen *ZA* array vectors.

B1.4.12.1 Z multi-vector operands

DPSTFY A multi-vector operand consisting of two or four SVE *Z* vector registers is called *Z* multi-vector operand.

RVCXBQ A *Z* multi-vector operand can occupy:

- Consecutively numbered *Z* registers.
- *Z* registers with strided numbering.

DNYNRZ A *Z* multi-vector operand occupying two consecutively numbered *Z* vectors consists of $Zn+0$ and $Zn+1$, where $n+x$ modulo 32 is a register number in the range 0 to 31 inclusive.

DDZDBM A *Z* multi-vector operand occupying four consecutively numbered *Z* vectors consists of $Zn+0$ to $Zn+3$, where $n+x$ modulo 32 is a register number in the range 0 to 31 inclusive.

DVYKCM The preferred disassembly for a *Z* multi-vector operand of consecutively numbered *Z* vectors is a dash-separated register range, for example { Z0.S-Z1.S } or { Z30.B-Z1.B }. Toolchains must also support assembler source code which uses

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the alternative comma-separated list notation, for example { Z0.S, Z1.S } or { Z30.B, Z31.B, Z0.B, Z1.B}, and disassemblers may provide an option to select between the dash-separated range and comma-separated list notations.

D_{PCYZS} A *Z* multi-vector operand occupying two *Z* vectors with strided register numbering consists of a first register in the range Z0-Z7 or Z16-Z23, followed by a second register with a number that is 8 higher than the first.

D_{RZTTV} A *Z* multi-vector operand occupying four *Z* vectors with strided register numbering consists of a first register in the range Z0-Z3 or Z16-Z19, followed by three registers with a number that is each 4 higher than the last.

D_{DMTSL} The preferred disassembly for a *Z* multi-vector operand of *Z* vectors with strided register numbering is a comma-separated register list, for example { Z0.D, Z8.D } or { Z0.H, Z4.H, Z8.H, Z12.H }.

B1.4.12.2 ZA multi-slice operands

D_{JMCTK} A multi-vector operand consisting of two or four *ZA* tile slices is called *ZA* multi-slice operand.

R_{SCHNH} A *ZA* multi-slice operand can occupy:

- Consecutively numbered horizontal *ZA* tile slices.
- Consecutively numbered vertical *ZA* tile slices.

D_{VXMGC} In a *ZA* multi-slice operand that consists of:

- Two *ZA* tile slices, the lowest numbered slice is a multiple of 2.
- Four *ZA* tile slices, the lowest numbered slice is a multiple of 4.

D_{JFDSB} In instructions operating on *ZA* multi-slice operands, the lowest-numbered slice is selected by rounding down the sum of a 32-bit general-purpose register (slice index register W*s*) and an offset (slice index offset *offs*) to the nearest lower multiple of the number of the slices in the ZA multi-slice operand, modulo the number of slices in the named tile.

R_{XMMKZ} Instructions operating on the following *ZA* multi-slice operands are treated as UNDEFINED:

- The four-slice operand in a 64-bit element tile when SVL is 128 bits.
- The two-slice operand in a 128-bit element tile when SVL is 128 bits.
- The four-slice operand in a 128-bit element tile when SVL is 128 bits or 256 bits.

D_{GJTMX} The preferred disassembly for a *ZA* multi-slice operand is as follows:

- ZA*t*H.*T*[W*s*, *offs1*:*offs2*], for horizontal *ZA* two-slice operands, where *offs2* = *offs1* + 1.
- ZA*t*H.*T*[W*s*, *offs1*:*offs4*], for horizontal *ZA* four-slice operands, where *offs4* = *offs1* + 3.
- ZA*t*V.*T*[W*s*, *offs1*:*offs2*], for vertical *ZA* two-slice operands, where *offs2* = *offs1* + 1.
- ZA*t*V.*T*[W*s*, *offs1*:*offs4*], for vertical *ZA* four-slice operands, where *offs4* = *offs1* + 3.

B1.4.12.3 ZA multi-vector operands

D_{RGXBK} A multi-vector operand consisting of two, four, eight, or sixteen *ZA* array vectors is called *ZA* multi-vector operand.

D_{TGDRF} One *ZA* array vector is called a *ZA* single-vector group.

D_{FCYGL} Two consecutively-numbered vectors in the *ZA* array are called a *ZA* double-vector group.

D_{GCTYB} Four consecutively-numbered vectors in the *ZA* array are called a *ZA* quad-vector group.

I_{PMQRQ} The *ZA* multi-vector operand consists of one, two, or four vector groups, where a vector group is one of the following:

- *ZA* single-vector group.
- *ZA* double-vector group.
- *ZA* quad-vector group.

I_{KLBYZ} The SME2 architecture includes multi-vector instructions that access a *ZA* multi-vector operand consisting of the same number of vector groups as there are vectors in each *Z* multi-vector operand.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{HPKZM} The preferred disassembly for a *ZA* multi-vector operand consisting of two or four vector groups, defined in declarations D_{KQZYZ}, D_{JWRSN}, and D_{TTNGH}, includes the symbol `VGx2` or `VGx4`, respectively. The symbol `VGx2` or `VGx4` can optionally be omitted in assembler source code if it can be inferred from the other operands.

D_{CLJBX} In instructions that access a *ZA* multi-vector operand, the lowest-numbered vector is selected by the sum of a 32-bit general-purpose register (vector select register W*v*) and an offset (vector select offset *offs*), modulo one of the following values:

- SVL_B when the operand consists of one *ZA* vector group.
- $SVL_B/2$ when the operand consists of two *ZA* vector groups.
- $SVL_B/4$ when the operand consists of four *ZA* vector groups.

B1.4.12.3.1 ZA multi-vector operands of single-vector groups

D_{QFPHH} In instructions where the *ZA* multi-vector operand consists of two single-vector groups, each vector group is held in a separate half of the *ZA* array: ZA[*n+0*] and ZA[$SVL_B/2$ + *n+0*], where *n* is in the range 0 to ($SVL_B/2$ - 1) inclusive.

D_{TTHGQ} In instructions where the *ZA* multi-vector operand consists of four single-vector groups, each vector group is held in a separate quarter of the *ZA* array: ZA[*n+0*], ZA[$SVL_B/4$ + *n+0*], ZA[$SVL_B/2$ + *n+0*], and ZA[SVL_B*3/4 + *n+0*], where *n* is in the range 0 to ($SVL_B/4$ - 1) inclusive.

D_{KQZYZ} The preferred disassembly for a *ZA* multi-vector operand of single-vector groups is:

- ZA.*T*[W*v*, *offs*, VGx2], when the operand consists of two single-vector groups.
- ZA.*T*[W*v*, *offs*, VGx4], when the operand consists of four single-vector groups.

Where *offs* is in the range 0 to 7 inclusive, and *T* is one of B, H, S, or D.

I_{BYBQLI} The mapping between *ZA* multi-vector operands of single-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-15 ZA array vector view and 32-bit element ZA tile slice view for single-vector groups

IMLNNG The mapping between *ZA* multi-vector operands of single-vector groups and 64-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-16 ZA array vector view and 64-bit element ZA tile slice view for single-vector groups

B1.4.12.3.2 ZA multi-vector operands of double-vector groups

D_{RBSQJ} In instructions where the *ZA* multi-vector operand consists of one double-vector group, the vector group is held in *ZA* array vectors: ZA[*n+0*] to ZA[*n+1*], where *n* is a multiple of 2 in the range 0 to (SVL_B - 2) inclusive.

D_{KKVVG} In instructions where the *ZA* multi-vector operand consists of two double-vector groups, each vector group is held in a separate half of the *ZA* array: ZA[*n+0*] to ZA[*n+1*], and ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+1*], where *n* is a multiple of 2 in the range 0 to (SVL_B/2 - 2) inclusive.

D_{VMYGN} In instructions where the *ZA* multi-vector operand consists of four double-vector groups, each vector group is held in a separate quarter of the *ZA* array: ZA[*n+0*] to ZA[*n+1*], ZA[SVL_B/4 + *n+0*] to ZA[SVL_B/4 + *n+1*], ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+1*], and ZA[SVL_B*3/4 + *n+0*] to ZA[SVL_B*3/4 + *n+1*], where *n* is a multiple of 2 in the range 0 to (SVL_B/4 - 2) inclusive.

D_{JWRSN} The preferred disassembly for a *ZA* multi-vector operand of double-vector groups is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ZA.*T*[W*v*, *offs1*:*offs2*], where: *offs1* is a multiple of 2 in the range 0 to 14 inclusive, when the operand consists of one double-vector group.
- ZA.*T*[W*v*, *offs1*:*offs2*, VGx2], where *offs1* is a multiple of 2 in the range 0 to 6 inclusive, when the operand consists of two double-vector groups.
- ZA.*T*[W*v*, *offs1*:*offs2*, VGx4], where *offs1* is a multiple of 2 in the range 0 to 6 inclusive, when the operand consists of four double-vector groups.

Where *offs2* = *offs1* + 1, and *T* is one of B, H, S, or D.

ILZRTK The mapping between *ZA* multi-vector operands of double-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Figure B1-17 ZA array vector view and 32-bit element ZA tile slice view for double-vector groups

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IJYQTB The mapping between *ZA* multi-vector operands of double-vector groups and 64-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Figure B1-18 ZA array vector view and 64-bit element ZA tile slice view for double-vector groups

B1.4.12.3.3 ZA multi-vector operands of quad-vector groups

DWSTWB In instructions where the *ZA* multi-vector operand consists of one quad-vector group, the vector group is held in *ZA* array vectors: ZA[*n*+*0*] to ZA[*n*+*3*], where *n* is a multiple of 4 in the range 0 to (SVL_B - 4) inclusive.

DQJXHS In instructions where the *ZA* multi-vector operand consists of two quad-vector groups, each vector group is held in a separate half of the *ZA* array: ZA[*n*+*0*] to ZA[*n*+*3*], and ZA[SVL_B/2 + *n*+*0*] to ZA[SVL_B/2 + *n*+*3*], where *n* is a

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

multiple of 4 in the range 0 to (SVL_B/2 - 4) inclusive.

D_{BQWJD} In instructions where the *ZA* multi-vector operand consists of four quad-vector groups, each vector group is held in a separate quarter of the *ZA* array: ZA[*n+0*] to ZA[*n+3*], ZA[SVL_B/4 + *n+0*] to ZA[SVL_B/4 + *n+3*], ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+3*], and ZA[SVL_B*3/4 + *n+0*] to ZA[SVL_B*3/4 + *n+3*], where *n* is a multiple of 4 in the range 0 to (SVL_B/4 - 4) inclusive.

D_{TTNGH} The preferred disassembly for a *ZA* multi-vector operand of quad-vector groups is:

- ZA.*T*[W*v*, *offs1*:*offs4*], where: *offs1* is a multiple of 4 in the range 0 to 12 inclusive, when the operand consists of one quad-vector group.
- ZA.*T*[W*v*, *offs1*:*offs4*, VGx2], where *offs1* is 0 or 4, when the operand consists of two quad-vector groups.
- ZA.*T*[W*v*, *offs1*:*offs4*, VGx4], where *offs1* is 0 or 4, when the operand consists of four quad-vector groups.

Where *offs4* = *offs1* + 3, and *T* is one of B, H, S, or D.

I_{ZNSGW} The mapping between *ZA* multi-vector operands of quad-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-19 ZA array vector view and 32-bit element ZA tile slice view for quad-vector groups

IKBMLX The mapping between *ZA* multi-vector operands of quad-vector groups and 64-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B1-20 ZA array vector view and 64-bit element ZA tile slice view for quad-vector groups

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4.13 SME2 ZT0 register

R_{KHZMV} A PE with FEAT_SME2 has a 512-bit register, *ZT0*, to store a lookup table.

R_{WDJBC} The ZT0 register holds 8-bit, 16-bit, or 32-bit lookup table elements that are stored in the least significant bits of 32-bit table entries. The lowest numbered 32 bits in the register hold table entry 0.

R_{JQXLS} The ZT0 register lookup table can be accessed using fully packed 2-bit or 4-bit indices from a numbered portion of one source Z*n* SVE scalable vector register.

I_{BRRGG} When the ZT0 register lookup table is addressed by 2-bit indices, four different table elements (0-3) of a given element size can be accessed. When the lookup table is addressed by 4-bit indices, 16 different table elements (0-15) of a given element size can be accessed.

R_{JKYRB} The indexed 8-bit, 16-bit, or 32-bit table elements are read from the ZT0 register and packed into consecutive or strided elements of an SVE Z vector or Z multi-vector operand.

See also:

- About PSTATE.ZA.
- Streaming SVE mode.
- Accessing PSTATE fields at EL0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.5 Software control features and EL0

The following sections describe the EL0 view of the software control features:

- Exception handling.
- Wait for Interrupt and Wait for Event.
- The YIELD instruction.
- Application level cache management.
- Instructions relating to Debug.
- About PSTATE.DIT.

B1.5.1 Exception handling

In the Arm architecture, an *exception* causes a change of program flow. Execution of an exception handler starts, at an Exception level higher than EL0, from a defined vector that relates to the exception taken.

Exceptions include:

- Interrupts.
- Memory system aborts.
- Exceptions generated by attempting to execute an instruction that is UNDEFINED.
- System calls.
- Secure monitor or Hypervisor traps.
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in The AArch64 System Level Programmers' Model.

The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a system call to an operating system.

The `BRK` instruction generates a Breakpoint Instruction exception. This provides a mechanism for debugging software using debugger executing on the same PE, see Breakpoint Instruction exceptions.

Note

The `BRK` instruction is supported only in the A64 instruction set. The equivalent instruction in the T32 and A32 instruction sets is `BKPT`.

B1.5.2 Wait for Interrupt and Wait for Event

Issuing a `WFI` instruction indicates that no further execution is required until a WFI wakeup event occurs, see Wait for Interrupt mechanism. This permits entry to a low-power state.

Issuing a `WFE` instruction indicates that no further execution is required until a WFE wakeup event occurs, see Wait for Event. This permits entry to a low-power state.

B1.5.3 The YIELD instruction

The `YIELD` instruction provides a hint that the task performed by a thread is of low importance so that it could yield, see YIELD. This mechanism can be used to improve overall performance in a *Symmetric Multithreading* (SMT) or *Symmetric Multiprocessing* (SMP) system.

Examples of when the `YIELD` instruction might be used include a thread that is sitting in a spin-lock, or where the arbitration priority of the snoop bit in an SMP system is modified. The `YIELD` instruction permits binary compatibility between SMT and SMP systems.

The `YIELD` instruction is a `NOP` hint instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The YIELD instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use for future migration to a multiprocessor or multithreading system. Operating systems can use YIELD in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

B1.5.4 Application level cache management

A small number of cache maintenance instructions can be enabled at EL0 from higher levels of privilege using the SCTLR_EL1 System register. Any access from EL0 to an operation without access permission causes the instruction to be trapped.

About the available operations, see Application level access to functionality related to caches.

B1.5.5 Instructions relating to Debug

Exception handling refers to the BRK instruction, which generates a Breakpoint Instruction exception. In addition, in AArch64 state and AArch32 state, the HLT instruction causes the PE to halt execution and enter Debug state. This provides a mechanism for debugging software using a debugger that is external to the PE, see About External Debug.

Note

In AArch32 state, previous versions of the architecture defined the DBG instruction, which could provide a hint to the debug system. This instruction executes as a NOP. Arm deprecates the use of the DBG instruction.

B1.5.6 About PSTATE.DIT

A *data-independent-time sequence of code* is the sequence of instructions executed from the first instruction executed after a change of PSTATE.DIT from 0 to 1 through the last instruction executed before a change of PSTATE.DIT from 1 to 0, inclusive. Instructions speculatively executed between the change of PSTATE.DIT from 0 to 1 and the change of PSTATE.DIT from 1 to 0 are considered part of the data-independent-time sequence of code.

A *data-independent-time resource* is any of:

- A general-purpose register other than the zero register (ZR)
- Stack Pointer register
- SIMD&FP register
- If FEAT_SVE or FEAT_SME is implemented, SVE scalable vector register
- If FEAT_SME is implemented, ZA register
- If FEAT_SME2 is implemented, ZT0 register
- SPSR_ELx.NZCV
- PSTATE.NZCV

Note

The collections of PSTATE.NZCV and SPSR_ELx.NZCV are each considered a single data-independent-time resource.

A *data-independent-time instruction* is an instruction that is required to honor the data-independent timing behaviors described in this section.

A *data-independent-time* value is any of:

- Any given value (V) read from memory by the Explicit Memory Read Effect of a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C), where V is not consumed by any other instruction (I2) for which any of the following apply:
  - I2 is not a data-independent-time instruction
  - I2 is not executed in C
  - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2
  - If I2 is an SME instruction, V is used as the slice index register or vector select register for I2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Any given value (V) written to a data-independent-time resource by a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C), where V is not consumed by any other instruction (I2) for which any of the following apply:
  - I2 is not a data-independent-time instruction
  - I2 is not executed in C
  - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2
  - If I2 is an SME instruction, V is used as the slice index register or vector select register for I2
- Any given value (V) read from a data-independent-time resource by a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C) and none of the following apply:
  - V was produced by another instruction (I0) and any of the following apply:
    - I0 is not a data-independent-time instruction
    - I0 is not executed in C
  - V is used to form the address or Logical Address Tag of a Memory Effect produced by I1.
  - V is not consumed by any other instruction (I2) for which any of the following apply:
    - I2 is not a data-independent-time instruction
    - I2 is not executed in C
    - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2
    - If I2 is an SME instruction, V is used as the slice index register or vector select register for I2
- Any given value (V) written to memory by the Explicit Memory Write Effect of a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C) and none of the following apply:
  - V was produced by another instruction (I0) and any of the following apply:
    - I0 is not a data-independent-time instruction
    - I0 is not executed in C
  - V is used to form the address or Logical Address Tag of a Memory Effect produced by I1.
  - V was sourced from the zero register.
  - V is not consumed by any other instruction (I2) for which any of the following apply:
    - I2 is not a data-independent-time instruction
    - I2 is not executed in C
    - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2
    - If I2 is an SME instruction, V is used as the slice index register or vector select register for I2
- Any given value (V) speculatively read from memory at a Location (L) as the result of executing a data-independent-time sequence of code, where L is not explicitly read by the data-independent-time sequence of code. Examples of this include:
  - Reading additional bytes from memory while reading the bytes required by an Explicit Memory Read Effect
  - Hardware-generated prefetches
  - Prefetch memory instructions

The execution time of a given data-independent-time sequence of code must be independent of all data-independent-time values.

The execution time of a data-independent-time instruction (I1) executed within a data-independent-time sequence of code must be independent of all data-independent-time values consumed or produced by I1.

The time to respond to an asynchronous exception taken from within a data-independent-time sequence of code must be independent of all data-independent-time values.

The execution of a data-independent-time sequence of code (C1) must not cause an irreversible change in microarchitectural state of the PE that would permit a sequence of code (C2) that is executed after C1 to determine any data-independent-time value accessed during the execution of C1. For the purposes of this requirement, a data-independent-time value written by an Explicit Memory Write Effect that is held in caches is not considered an irreversible change in microarchitectural state.

The execution of a data-independent-time sequence of code (C1) must not alter the state of PMU counters in a way that allows a sequence of code (C2) that is executed after C1 to determine any data-independent-time value accessed during C1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In cases where an instruction is architecturally required to only read or write a subset of a data-independent-time resource, the entire data-independent-time resource is considered to have been read or written for the sake of these requirements.

Note

The architecture does not guarantee that the execution time of a data-independent-time sequence of code is independent of data-independent-time values if those values are read-from or written-to a peripheral.

Note

The classification of a value being a data-independent-time value due to having been read from a memory location (L) by an Explicit Memory Read Effect or written to L by an Explicit Memory Write Effect is based on the particular Effect generated by a data-independent-time instruction executed from within a data-independent-time sequence of code. The value held in L is not itself considered a data-independent-time value. As a result, it is possible for the execution time of a data-independent-time sequence of code to vary based on the value held in L if L is accessed via any of the following:

- Outside the data-independent-time sequence of code; either architecturally or speculatively
- Within a data-independent-time sequence of code but not as a data-independent-time value

Note

- The *Operational information* section of an instruction description indicates whether or not that instruction is a data-independent-time instruction. If the *Operational information* section of an instruction description does not mention data-independent-time or if the section does not exist, then the instruction is not a data-independent-time instruction.
- The use of value prediction for data-independent-time values, is not compatible with the requirement that the timing is insensitive to the data value being loaded.
- Arm recommends that the FEAT_PAuth instructions do not have their timing dependent on the key value used in the pointer authentication, regardless of whether the instruction is executed in a data-independent-time sequence of code.
- The architecture makes no statement about the timing properties of any instructions executed outside of a data-independent-time sequence of code. However, it is likely that these instructions have timing that is invariant of the data in many situations.
- For SVE and SME predicated instructions, it is the programmer's responsibility to use a *Governing predicate* that does not reflect the values of the data being operated on.

A corresponding DIT bit is added to PSTATE in AArch64 state, and to CPSR in AArch32 state.

On an exception that is taken from AArch64 state to AArch64 state, PSTATE.DIT is copied to SPSR_ELx.DIT.

On an exception that is taken from AArch32 state to AArch64 state, CPSR.DIT is copied to SPSR_ELx.DIT.

On an exception return from AArch64 state:

- SPSR_ELx.DIT is copied to PSTATE.DIT, when the target Exception level is in AArch64 state.
- SPSR_ELx.DIT is copied to CPSR.DIT, when the target Exception level is in AArch32 state.

PSTATE.DIT can be written and read at all Exception levels.

Note

- PSTATE.DIT is unchanged on entry into Debug state.
- PSTATE.DIT is not guaranteed to have any effect in Debug state.
- Software is advised to ensure that no indirect branch is executed while secret data is held in registers.
- Software is advised to ensure that no direct branches target non-constant-time code while secret data is held in registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B2
The AArch64 Application Level Memory Model

This chapter gives an application level view of the memory model. It contains the following sections:

- About the Arm memory model.
- Atomicity in the Arm architecture.
- Ordering requirements defined by the formal concurrency model.
- Additional ordering requirements outside of the scope of the formal concurrency model
- Restrictions on the effects of speculation.
- Memory barriers.
- Caches and memory hierarchy.
- Alignment support.
- Endian support.
- Memory types and attributes.
- Mismatched memory attributes.
- Synchronization and semaphores.

Note

In this chapter, System register names usually link to the description of the register in AArch64 System Register Descriptions, for example. SCTLR_EL1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the memory model.

This section contains:

- Address space.
- Memory type overview.
- SVE memory model.
- SME memory model.

B2.1.1 Address space

Address calculations are performed using 64-bit registers.

However, supervisory software can configure the top eight address bits for use as a tag, as described in Address tagging. If this is done, address bits[63:56]:

- Are not considered when determining whether the address is valid.
- Are never propagated to the Program Counter.

Supervisory software determines the valid address range. Attempting to access an address that is not valid generates an MMU fault.

Simple sequential execution of instructions might overflow the valid address range. For more information, see Virtual address space overflow.

Memory accesses use the `Mem[]` function. This function makes an access of the required type. If supervisory software configures the top eight address bits for use as a tag, the top eight address bits are ignored.

The `AccessType` defines the different access types and attributes.

Note

- The AArch64 System Level Memory Model and The AArch64 Virtual Memory System Architecture include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also include pseudocode descriptions of these operations.
- For information on the pseudocode that relates to memory accesses, see Basic memory access, Unaligned memory access, and Aligned memory access.

B2.1.2 Memory type overview

The Arm architecture provides the following mutually-exclusive memory types:

Normal This is generally used for the majority of memory operations, both read/write and read-only operations.

Device The Arm architecture forbids Speculative reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive Locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See Gathering.
- They preserve the access order and synchronization requirements for accesses to a single peripheral. See Reordering.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- They indicate whether a write can be acknowledged other than at the end point. See Early Write Acknowledgement.

For more information on Normal memory and Device memory, see Memory types and attributes.

Note

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-ordered memory type. A *Note* in Device memory describes how these memory types map onto the Armv8 memory types.

B2.1.3 SVE memory model

I_{CZFSY} SVE predicated memory operations have a vector element size and a memory element access size. The vector element size specifies the data that is read from and written to the vector. The memory element access size specifies the amount of data that is read from and written to the memory.

I_{TJQJF} The vector element size and the memory element access size do not need to have the same value.

I_{LGGHH} For each memory element, there is an associated element address.

SVE also affects behavior in the following areas:

- Requirements for single-copy atomicity.
- SVE memory ordering relaxations.
- Load or Store of Single or Multiple registers.
- Endianness in SVE operations.
- SVE loads and stores that access Device memory.

B2.1.4 SME memory model

R_{BQSCG} SME load/store memory accesses are subject to the same rules that govern SVE load/store memory accesses.

I_{WWVYJ} When the PE is in *Streaming SVE mode*, there are relaxations for Advanced SIMD&FP instructions in the following areas:

- Streaming SVE mode memory ordering relaxations.
- Streaming SVE mode loads and stores that access Device memory.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The Arm architecture description refers to two types of atomicity, *single-copy atomicity* and *multi-copy atomicity*. In the Arm architecture, the atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information, see:

- Requirements for single-copy atomicity.
- Properties of single-copy atomic accesses.
- Multi-copy atomicity.
- Requirements for multi-copy atomicity.
- Concurrent modification and execution of instructions.
- Possible implementation restrictions on using atomic instructions.

For more information about the memory types, see Memory type overview.

B2.2.1 Requirements for single-copy atomicity

For explicit memory effects generated from an Exception level the following rules apply:

- A read that is generated by a load instruction that loads a single general-purpose register and is aligned to the size of the read in the instruction is single-copy atomic.
- A write that is generated by a store instruction that stores a single general-purpose register and is aligned to the size of the write in the instruction is single-copy atomic.
- Reads that are generated by a Load Pair instruction that loads two general-purpose registers and are aligned to the size of the load to each register are treated as two single-copy atomic reads, one for each register being loaded.
- Writes that are generated by a Store pair instruction that stores two general-purpose registers and are aligned to the size of the store of each register are treated as two single-copy atomic writes, one for each register being stored.
- Load-Exclusive Pair instructions of two 32-bit quantities and Store-Exclusive Pair instructions of 32-bit quantities are single-copy atomic.
- When the Store-Exclusive of a Load-Exclusive/Store-Exclusive pair instruction using two 64-bit quantities succeeds, it causes a single-copy atomic update of the entire memory location being updated.

  Note

  To atomically load two 64-bit quantities, perform a Load-Exclusive pair/Store-Exclusive pair sequence of reading and writing the same value for which the Store-Exclusive pair succeeds, and use the read values from the Load-Exclusive pair.

- Where translation table walks generate a read of a translation table entry, this read is single-copy atomic.
- For the atomicity of instruction fetches, see Concurrent modification and execution of instructions.
- Reads to SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being loaded are treated as single-copy atomic reads.
- Writes from SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being stored are treated as single-copy atomic writes.
- Element or Structure Reads to SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being loaded, have each element treated as a single-copy atomic read.
- Element or Structure Writes from SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being stored, have each element treated as a single-copy atomic store.
- Reads to SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit reads.
- Writes from SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit writes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_LS64WB is implemented, accesses to two 128-bit SIMD and floating-point registers by an LDP (SIMD&FP) or STP (SIMD&FP) instruction are treated as one single-copy atomic 256-bit access when both the following are true:
  - The access is 32-byte aligned in memory.
  - The access is to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory.
- SVE predicated load and store instructions have the same single-copy atomicity guarantees as the SIMD load and store instructions for 64-bit or smaller elements.
- An SVE predicated load or store of a 128-bit element that is 64-bit aligned is treated as a pair of 64-bit single-copy atomic accesses.
- SVE unpredicated load and store instructions are performed as a sequence of byte accesses.
- SVE unpredicated load and store instructions do not guarantee that any access larger than a byte will be performed as a single-copy atomic access.
- When FEAT_LS64 is implemented, a single-copy atomic load of a 64-byte value that is 64-byte aligned in memory is treated as an atomic 64-byte read from the target address.
- When FEAT_LS64 is implemented, a single-copy atomic store of a 64-byte value that is 64-byte aligned in memory is treated as an atomic 64-byte write to the target address.
- For unaligned memory accesses, the single-copy atomicity is described in Alignment of data accesses.
- The read and write generated by an Atomic instruction is single-copy atomic at the overall size of the data access.

All other memory accesses are regarded as streams of accesses to bytes, and no atomicity between accesses to different bytes is ensured by the architecture.

All accesses to any byte are single-copy atomic.

Note

In AArch64 state, no memory accesses from a `DC ZVA` have single-copy atomicity of any quantity greater than individual bytes.

If, according to these rules, an instruction is executed as a sequence of accesses, exceptions, including interrupts, can be taken during that sequence, regardless of the memory type being accessed. If any of these exceptions are returned from using their preferred return address, the instruction that generated the sequence of accesses is re-executed, and so any access performed before the exception was taken is repeated. See also Taking an asynchronous exception during a multi-access load or store.

Note

The exception behavior for these multiple access instructions means that they are not suitable for use for writes to memory for the purpose of software synchronization.

B2.2.1.1 Changes to single-copy atomicity in Armv8.4

In addition to the single-copy atomicity requirements listed above:

Instructions that are introduced in FEAT_LRCPC are single-copy atomic when all of the following conditions are true:

- All bytes being accessed are within the same 16-byte quantity aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, all loads and stores are single-copy atomic when all of the following conditions are true:

- Accesses are unaligned to their data size but all bytes being accessed are within a 16-byte quantity that is aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, `LDP`, `LDNP`, and `STP` instructions that load or store two 64-bit registers are single-copy atomic when all of the following conditions are true:

- The overall memory access is aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_LSE2 is implemented, `LDP`, `LDNP`, and `STP` instructions that access fewer than 16 bytes are single-copy atomic when all of the following conditions are true:

- All bytes being accessed are within a 16-byte quantity aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

B2.2.2 Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.
2. For a single-copy atomic load instruction L_1 that overlaps a single-copy atomic store instruction S_2, if one of the overlapping reads generated by L_1 Reads-from one of the overlapping writes generated by S_2, then none of the overlapping writes generated by S_2 are Coherence-after the corresponding overlapping reads generated by L_1.

For more information, see Ordering requirements defined by the formal concurrency model.

B2.2.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Note

Writes that are not coherent are not multi-copy atomic.

B2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The Arm memory model is Other-multi-copy atomic. For more information, see External visibility requirement.

B2.2.5 Concurrent modification and execution of instructions (CMODX)

The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction behaving like any possible sequence of instructions executable at the same Exception level. However, when both the instruction before and the instruction after modification is one of `B`, `B.cond`, `BL`, `BRK`, `CB<cc>`, `CBB<cc>`, `CBH<cc>`, `CBNZ`, `CBZ`, `HVC`, `ISB`, `NOP`, `SMC`, `SVC`, `TBNZ`, `TBZ`, or `TRCIT`, the architecture guarantees that after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn                 ; Clean data cache by VA to point of unification (PoU)
        DSB ISH                     ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn                 ; Invalidate instruction cache by VA to PoU
        DSB ISH
```

Note

- The `DC CVAU` operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
- If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- In a multiprocessor system, the DC CVAU and IC IVAU are broadcast to all PEs within the Inner Shareable domain of the PE running this sequence.

3. When the modified instructions are observable, each PE that is executing the modified instructions must execute an `ISB` or perform a context synchronization event to ensure execution of the modified instructions:

```
        ISB                             ; Synchronize fetched instruction stream
```

For more information about the required synchronization operation, see Synchronization and coherency issues between data and instruction accesses.

For information about memory accesses caused by instruction fetches, see Ordering relations.

B2.2.6 Possible implementation restrictions on using atomic instructions

In some implementations, and for some memory types, the properties of atomicity can be met only by functionality outside the PE. Some system implementations might not support atomic instructions for all regions of the memory. In particular, this can apply to:

- Any type of memory in the system that does not support hardware cache coherency.
- Device, Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such implementations, it is defined by the system:

- Whether the atomic instructions are atomic in regard to other agents that access memory.
- If the atomic instructions are atomic in regard to other agents that access memory, which address ranges or memory types this applies to.

An implementation can choose which memory type is treated as Non-cacheable.

The memory types for which it is architecturally guaranteed that the atomic instructions will be atomic are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

The architecture only requires that Conventional memory that is mapped in this way supports this functionality.

If the atomic instructions are not atomic in regard to other agents that access memory, then performing an atomic instruction to such a location can have one or more of the following effects:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction generates a synchronous External abort.
- The instruction generates a System Error interrupt.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

  For the EL1&0 translation regime, if the atomic instruction is not supported because of the memory type that is defined in the first stage of translation, or the second stage of translation is not enabled, then this exception is a first stage abort and is taken to EL1. Otherwise, the exception is a second stage abort and is taken to EL2.
- The instruction is treated as a `NOP`.
- The instructions are performed, but there is no guarantee that the memory accesses were performed atomically in regard to other agents that access memory. In this case, the instruction might also generate a System Error interrupt.

A system implementation might support atomic floating-point memory operation instructions on different regions of Device and Non-cacheable memory from those supported by other atomic instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3 Ordering requirements defined by the formal concurrency model

The material presented in the current section corresponds to what is described in the formal concurrency model. The definitions of relations, including but not limited to Dependency-ordered-before, Barrier-ordered-before, or Atomic-ordered-before, are automatically transliterated from the formal concurrency model.

The Arm memory model introduces a number of relations which are as follows:

- *Intrinsic relations*; For example, Intrinsic Data Dependencies, Intrinsic Control Dependencies, Intrinsic Order Dependencies and Translation-intrinsically-before are hardware requirements which stem from the instruction semantics.
- *After relations*; For example, Coherence-after and TLBI-after are relations which happen to be oriented that way in a specific execution, but could be oriented the other way in a different execution.
- *Observation relations*; For example, Explicit-Observed-by and TLBI-Observed-by build on *After relations* to describe an execution.
- *Ordered relations*; For example, Ordered-before and TLBI-ordered-before are architectural requirements which must be respected by hardware in all executions.

This section describes observation and ordering in the Arm memory model. It contains the following subsections:

- Basic definitions.
- Intrinsic Dependency relations.
- Tag-check-intrinsically-before.
- Translation-intrinsically-before.
- Fetch-intrinsically-before.
- Dependency relations.
- Ordering relations.
- Observation relations.
- Definition of an Architecturally Allowed Execution.
- Alternative formulations of the External visibility requirement for Explicit Memory Effects.

For more information about endpoint ordering of memory accesses, see Reordering.

In the Arm memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers. See Memory types and attributes.

The Arm architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:

- The AArch64 System Level Memory Model.
- The AArch64 Virtual Memory System Architecture.

See also Mismatched memory attributes.

For the purposes of the memory model, all Effects generated due to execution of an instruction are considered to be generated by the PE on which the instruction is executed. The list of all Effects of an instruction in the scope of the Arm Memory Model is further presented in Effects.

In this section Ordering requirements defined by the formal concurrency model, all of the following are assumed:

- One of the following applies:
  - All PEs are part of the same Inner Shareable domain and all Memory Effects are to Inner Shareable or Outer Shareable memory as long as no barriers or TLB invalidations specify the Non-shareable domain.
  - All PEs are part of the same Outer Shareable domain and all Memory Effects are to Outer Shareable memory as long as no barriers or TLB invalidations specify the Non-shareable domain or the Inner Shareable domain.
  - All PEs are part of the Full System domain and all Memory Effects are to Normal, Inner Non-Cacheable, Outer Non-Cacheable memory or Device memory as long as all barriers specify the Full System domain and there are no TLB invalidations.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All Memory Effects are to Normal, Inner Write-Back, Outer Write-Back memory. Behaviors in Ordering requirements defined by the formal concurrency model also apply to Memory Effects to memory with attributes other than Normal, Inner Write-Back, Outer Write-Back (for example, Device or Inner Write-Through, Outer Write-Through) as long as the behavior of the instruction which generated the Memory Effect is guaranteed for the memory type. For example, atomic instructions are only guaranteed to be atomic for Normal, Inner Write-Back, Outer Write-Back, Shareable memory. Memory Effects to memory with attributes other than Normal, Inner Write-Back, Outer Write-Back may require additional ordering behaviors than specified in Ordering requirements defined by the formal concurrency model. Such additional behaviors are described in Memory types and attributes.
- All Memory Effects to a given Location are not using mismatched memory attributes as described in Mismatched memory attributes.

B2.3.1 Basic definitions

The Arm memory model provides a set of definitions that are used to construct conditions on the permitted sequences of accesses to memory.

Location

A *Location* is a Memory Location or a Tag Location. A *Memory Location* is a byte of data that is associated with an address in the physical address space. A *Tag Location* is a 4-bit MTE Allocation Tag that is associated with an address in the physical address space.

Effects

The *Effects* of an instruction can be:

- Register Effects.
- Memory Effects.
- Explicit Memory Effects.
- Implicit Translation Table Descriptor (TTD) Memory Effects.
- Hardware Update Effects.
- Tag Memory Effects.
- Implicit Instruction Memory Read Effects.
- Barrier Effects.
- Instruction Fetch Barrier Effects.
- Conditional Branching, and Intrinsic Branching Effects.
- Fault Effects, Exception Entry, and Exception Return Effects.
- TLBI Effects, Completed TLBI Effects and Invalidation Scopes.
- TLBUncacheable Effects.
- DC CVAU Effects.
- IC Effects, Completed IC Effects.
- Empty Effects.

Program order

An Effect E_1 of an instruction I_1 appears in program order before an Effect E_2 of an instruction I_2 if and only if I_1 occurs before I_2 in the order specified by the program. Each Effect generated by an instruction has a unique Effect identifier that characterizes it among the Effects generated by the same instruction.

Register Effects

The *Register Effects* of an instruction are register reads or register writes of that instruction. Register Effects only pertain to:

- General purpose registers from R0 to R30, excluding the zero register.
- SVE registers.
- SIMD/FP registers.
- PSTATE.NZCV.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Direct reads from System registers that are readable.
- Direct writes to System registers that are writable.
- Direct reads from Special-purpose registers that are readable.
- Direct writes to Special-purpose registers that are writable.

For an instruction that accesses registers, a *Register Read Effect* is generated for each register read by the instruction and a *Register Write Effect* is generated for each register written by the instruction. An instruction may generate both Register Read and Write Effects.

Memory Effects

The *Memory Effects* of an instruction are the Read or Write Effects from memory of that instruction. For an instruction that accesses memory, a *Memory Read Effect* is generated for each Location read from by the instruction and a *Memory Write Effect* is generated for each Location written to by the instruction. An instruction can generate both Memory Read and Write Effects.

Explicit Memory Effects

An *Explicit Memory Effect* is a Memory Read or Memory Write Effect that is generated by one of the following:

- A load instruction, store instruction, atomic instruction, Tag and Data store instruction, `DC ZVA`, `DC GZVA`, or `STZGM`; to access the Memory Location(s) addressed by any of the following from the instruction:
  - The `Xn|SP` operand, plus any applicable offset.
  - The `PC` plus the offset specified by the `<label>` argument.
- A Tag load instruction, Tag store instruction, Tag and Data store instruction, Bulk Allocation tag access instruction, `DC GZVA`, or `DC GVA`; to access the Tag Location(s) addressed by the `Xn|SP` operand, plus any applicable offset, of the instruction.
- A Memory Copy or Memory Set instruction; to access the Memory Location(s) addressed by the `Xs` or `Xd` operand of the instruction, including all of the bytes within the range specified by the `Xn` operand of the instruction.
- A Memory Set with tag setting instruction; to access the Tag Location(s) addressed by the `Xs` or `Xd` operand of the instruction, including all of the Allocation tags within the range specified by the `Xn` operand of the instruction.
- An `MRS` or `MSR` instruction that is transformed into a Memory Effect due to FEAT_NV2; to access the Memory Locations addressed by VNCR_EL2 plus the offset for the specified register.

Note

Prefetch memory instructions do not generate any Explicit Memory Effects.

Implicit Translation Table Descriptor Memory Effects

Implicit Translation Table Descriptor Memory Effects (Implicit TTD Memory Effects) are Memory Read or Write Effects to translation table descriptors caused by address translation, including the hardware updates for the Access Flag or the Dirty State of a TTD.

Hardware Update Effects

Hardware Update Effects are Implicit TTD Memory Write Effects that correspond to writes modifying the Access Flag or the Dirty State of a TTD.

For more information on Hardware Updates of the Access Flag or Dirty State, see Hardware management of the Access flag and Hardware management of the dirty state.

Note

A Hardware Update Effect for the Access Flag may occur even in the absence of corresponding architecturally executed instructions, in which case the Hardware Update Effect is considered autonomous. Autonomous Hardware Update Effects are not constrained by the ordering relations that apply to the Hardware Update Effects of architecturally executed instructions.

Tag Memory Effects

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Tag Memory Effects are a subset of Memory Effects. A Tag Memory Effect refers to a Memory Read or Write Effect to a Tag Location. For example:

- `LDG` and `LDGM` instructions generate an Explicit Tag Memory Read Effect for their read from a Tag Location.
- `STG`, `STGM`, `STGP`, `DC GZVA`, and `DC GVA` instructions generate an Explicit Tag Memory Write Effect for their write to a Tag Location.
- A Tag Checked memory access generates an Implicit Tag Memory Read Effect for its read of a Tag Location, as well as an Explicit Data Memory Effect.

Implicit Instruction Memory Read Effects

An *Implicit Instruction Memory Read Effect* refers to a Memory Read Effect generated as a result of an instruction fetch.

Barrier Effects

Barrier instructions generate *Barrier Effects*. In this chapter, the resulting Effect is named after the instruction. Therefore, an ISB instruction generates an ISB Effect, a DMB instruction generates a DMB Effect, and a DSB instruction generates a DSB Effect.

The following conventions are used when referring to DMB Effects:

- A DMB FULL Effect is generated by a DMB instruction with neither a LD nor ST qualifier.
- A DMB LD Effect is generated by a DMB instruction with the LD qualifier.
- A DMB ST Effect is generated by a DMB instruction with the ST qualifier.

The following conventions are used when referring to DSB Effects:

- A DSB FULL Effect is generated by a DSB instruction with neither a LD nor ST qualifier.
- A DSB LD Effect is generated by a DSB instruction with the LD qualifier.
- A DSB ST Effect is generated by a DSB instruction with the ST qualifier.

Instruction Fetch Barrier Effects

An *Instruction Fetch Barrier Effect* (IFBE) is one of:

- A Context Synchronization event.
- An exception entry to ELx, regardless of the value of SCTLR_ELx.EIS, due to an exception generated for any reason other than any of the following:
  - `SVC` instruction execution that is not trapped.
  - `HVC` instruction execution that is not disabled.
  - `SMC` instruction execution that is not trapped or disabled.
  - `BKPT` instruction execution.
  - `BRK` instruction execution.

Note

- If FEAT_ExS is not implemented, or if FEAT_ExS is implemented and the SCTLR_ELx.EOS field is 1, an Exception Return effect is an IFBE due to it being a Context Synchronization event.
- If FEAT_ExS is not implemented, or if FEAT_ExS is implemented and the SCTLR_ELx.EIS field is 1, the Exception entry effects due to the exceptions excluded above are IFBEs due to being Context Synchronization events.

Conditional Branching, and Intrinsic Branching Effects

Branching Effects are generated when a choice is made between two possible paths in the execution flow. Branching Effects represent such decision points.

Conditional Branching Effects are generated by Conditional branch instructions, when as a consequence of evaluating a condition a choice potentially affecting the execution flow of other subsequent instructions is made.

Intrinsic Branching Effects due to a translation occur when, as a consequence of reading a TTD, a choice is made between raising a fault (for example, if the descriptor is invalid) or performing a physical access. *Intrinsic Branching Effects* represent such decision points.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Intrinsic Branching Effects due to tag checking occur when, as a consequence of reading a tag, a choice is made between raising a fault (that is, due to a failed tag check) or performing a physical access. *Intrinsic Branching Effects* represent such decision points.

Intrinsic Branching Effects due to instruction fetch occur when, as a consequence of reading and decoding an instruction, a choice is made between raising a fault (for example, as a result of fetching an illegal instruction) or proceeding to execute the valid instruction. *Intrinsic Branching Effects* represent such decision points.

Note

Conditional Selection instructions and Compare-and-Swap instructions generate *Intrinsic Branching Effects*.

Fault Effects, Exception Entry, and Exception Return Effects

An instruction that encounters a fault generates a Fault Effect. An instruction that encounters a fault leading to a synchronous exception generates a Fault Effect that is also an *Exception Entry Effect*, for example as a result of reading an invalid TTD or fetching an illegal instruction.

An ERET instruction generates an *Exception Return Effect.*

A Fault Effect generated by the MMU is called an *MMU Fault Effect*. See MMU fault types for a full list of faults generated by the MMU.

A Fault Effect generated by a failed Tag Check is called a *TagCheck Fault Effect*.

Instructions that write to memory with Release semantics generate Fault Effects with Release semantics when they encounter a fault.

Note

- An instruction that encounters an asynchronous Tag Check Fault generates a TagCheck Fault Effect that is not an Exception Entry Effect.
- An instruction that encounters a synchronous Tag Check Fault generates a TagCheck Fault Effect that is also an Exception Entry Effect.

TLBI Effects, Completed TLBI Effects and Invalidation Scopes

TLBI Effects are generated by TLBI instructions.

A TLBI Effect E_1 is a *Completed TLBI Effect* if E_1 appears in program order before a DSB FULL Effect.

The *Invalidation scope* of a TLBI Effect is a set Memory Effects. For a Virtual Address *x*, a Memory Read or Write Effect to the TTD of x is in the Invalidation Scope of a TLBI Effect if *x* is in the scope of the TLBI instruction. For the full description of the scope of TLBI instructions, see TLB maintenance instruction scope.

TLBUncacheable Effects

An MMU Fault Effect is TLBUncacheable if it is one of:

- A Translation Fault Effect.
- An Access Flag Fault Effect.
- An Address Size Fault Effect.
- A synchronous External Abort on translation table walk, such that it cannot be determined whether the TTD would otherwise generate a Translation Fault, Access Flag Fault, or Address Size Fault effect.

  Note

  A Permission Fault Effect is not TLBUncacheable.

DC CVAU Effects

DC CVAU Effects are generated by DC CVAU instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IC Effects, Completed IC Effects

IC Effects are generated by IC IALLUIS, IC IALLU or IC IVAU instructions. An IC Effect E_1 is a *Completed IC Effect* if E_1 appears in program order before a DSB FULL Effect.

Empty Effects

Empty Effects are generated by instructions where a decision point, represented by a Branching Effect, leads to no further action resulting from that decision. For example, when FEAT_MTE_ASYNC is implemented and Tag Check Faults are reported asynchronously, a successful tag comparison generates an Empty Effect instead of a TagCheck Fault Effect.

Low-order Bits

The Low-order Bits of an address are the bits that remain unchanged in the translation process of an input address to an output address. The number of low-order bits that remain unchanged depends on the translation granule size.

Same Low-order Bits

Two Effects E_1 and E_2 have the Same Low-order Bits if the Low-order Bits of the address of E_1 and the Low-order Bits of the address of E_2 match.

Note

The address associated with an MMU Fault Effect is the faulting address as indicated in FAR_ELx.

Effects with a valid Physical Address

An Effect E_1 has a valid Physical Address if any of the following applies:

- All of the following apply:
  - At least one stage of address translation is enabled.
  - Any of the following applies:
    - E_1 is an Implicit TTD Memory Effect and the Physical Address of E_1 does not generate an Address Size fault.
    - All of the following apply:
      - E_2 is Translation-intrinsically-before E_1.
      - E_2 is an Implicit TTD Memory Read Effect associated with the final stage of translation that has a valid Output Address.
- All of the following apply:
  - Address translation is not enabled.
  - One of the following applies:
    - E_1 is a Memory Effect.
    - E_1 is a DC Effect.
    - E_1 is a IC Effect.
  - The Physical Address of E_1 does not generate an Address Size Fault.

Note

In situations where a memory access to a Physical Address generates an Address Size Fault, the PE does not generate the corresponding Memory Effect or DC Effect or IC Effect.

Same Location

An Effect E_1 and an Effect E_2 are to the *Same Location* if one of the following applies:

- All of the following apply:
  - E_1 has a valid Physical Address.
  - It is not the case that E_1 is a Tag Memory Effect.
  - E_1 and E_2 are to the same Physical Address.
  - E_2 has a valid Physical Address.
  - It is not the case that E_2 is a Tag Memory Effect.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All of the following apply:
  - One of the following applies:
    - E_1 is a Tag Memory Effect.
    - E_1 is an MMU Fault Effect that has a valid Physical Address.
  - E_1 and E_2 are to the same Physical Address.
  - One of the following applies:
    - E_2 is a Tag Memory Effect.
    - E_2 is an MMU Fault Effect that has a valid Physical Address.
- All of the following apply:
  - E_1 is an MMU Fault Effect that does not have a valid Physical Address.
  - E_1 and E_2 have the Same Low Order Bits.
- All of the following apply:
  - E_1 and E_2 have the Same Low Order Bits.
  - E_2 is an MMU Fault Effect that does not have a valid Physical Address.

Note

An Effect E_1 and an Effect E_2 are to the Same Location, if E_1 and E_2 are Memory Effects or Fault Effects to the same PA. The PA for a Fault Effect that prevents an output address from being generated, for example a Translation Fault Effect, is considered to be the same as the PA of an Explicit Memory Effect if the two Effects have the Same Low-order Bits.

Same Cache Line

An Effect E_1 and an Effect E_2 are to the *Same Cache Line* if:

- E_1 and E_2 are Memory Effects or DC CVAU Effects or IC Effects.
- E_1 and E_2 are to PAs that are on the Same Cache Line as follows from section Cache identification.

Successful Read-Modify-Write pair

Two Effects E_1 and E_2 form a *Successful Read-Modify-Write* pair if all of the following apply:

- E_1 is a Memory Read Effect.
- E_2 is a Memory Write Effect.
- E_1 and E_2 are to the Same Location.
- One of the following applies:
  - E_1 and E_2 are generated by the same atomic instruction.
  - E_1 and E_2 are generated by the same successful Load-Exclusive/Store-Exclusive instruction pair.

Reads-from-memory

The *Reads-from-memory* relation couples Memory Read and Write Effects to the same Location so that each Memory Read Effect is paired with exactly one Memory Write Effect in the execution of a program. A Memory Read Effect E_2 Reads-from-memory a Memory Write Effect E_1, if and only if E_1 and E_2 are to the same location and E_2 takes its data from E_1.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_1 Reads-from-memory E_2.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_1 and E_2 form a successful Read-Modify-Write pair.

then it is not the case that E_1 Reads-from-memory E_2.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Implicit Tag Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_1 Reads-from-memory E_2.

Coherence order

There is a per-location *Coherence order* relation that provides a total order over all Memory Write Effects to that Location, starting with a notional Memory Write Effect of the initial value. The Coherence order of a Location represents the order in which Memory Write Effects to the Location arrive at memory.

Local register read successor

An Effect E_2 is a *Local register read successor* of an Effect E_1 if all of the following apply:

- E_1 is a Register Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the same Register.
- It is not the case that all of the following apply:
  - E_1 appears in program order before E_3.
  - E_1 and E_3 are to the same Register.
  - E_3 is a Register Write Effect.
  - E_3 appears in program order before E_2.
  - E_3 and E_2 are to the same Register.
- E_2 is a Register Read Effect.

Local memory read successor

An Effect E_2 is a *Local memory read successor* of an Effect E_1 if all of the following apply:

- E_1 is a Memory Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.
- E_2 is a Memory Read Effect.
- There is no Memory Write Effect E_3 to the Same Location as E_1 such that E_1 appears in program order before E_3 and E_3 appears in program order before E_2.

Local memory write successor

An Effect E_2 is a *Local memory write successor* of an Effect E_1 if one of the following applies:

- All of the following apply:
  - One of the following applies:
    - E_1 is an Explicit Memory Effect.
    - E_1 is an Implicit Tag Memory Read Effect.
  - E_1 appears in program order before E_2.
  - E_1 and E_2 are to the Same Location.
  - E_2 is an Explicit Memory Write Effect.
- All of the following apply:
  - E_1 is an Implicit TTD Memory Read Effect.
  - E_1 appears in program order before E_2.
  - E_1 and E_2 are to the Same Location.
  - One of the following applies:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— E_2 is an Explicit Memory Write Effect.
— E_2 is a Hardware Update Effect.

Local MMU Fault successor

An Effect E_2 is a *Local MMU Fault successor* of an Effect E_1 if all of the following apply:

- E_1 is an Explicit Memory Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.
- E_2 is an MMU Fault Effect.

Coherence-before, Coherence-after

A Memory Write Effect E_1 is *Coherence-before* a Memory Write Effect E_2 to the same Location if E_1 is sequenced before E_2 in the Coherence order for the Location.

A Memory Read Effect E_1 is *Coherence-before* a Memory Write Effect E_2 to the same Location if E_1 Reads-from-memory a Memory Write Effect E_3 and E_3 is Coherence-before E_2.

An Effect E_2 is *Coherence-after* an Effect E_1 if E_1 is *Coherence-before* E_2.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Explicit Memory Read Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Implicit Tag Memory Read Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1.

TLBI-before, TLBI-after

For two Effects E_1 and E_2, if and only if all of the following applies:

- E_1 is a Completed TLBI Effect.
- E_2 is an Implicit TTD Memory Read Effect.
- E_2 is in the Invalidation Scope of E_1.

then one and only one of the following applies:

- E_1 is *TLBI-after* E_2 (equivalently E_2 is *TLBI-before* E_1), or
- E_2 is *TLBI-after* E_1 (equivalently E_1 is *TLBI-before* E_2).

Note

The TLBI-after relation enumerates all possible pairs (E_1,E_2) where E_1 is a TLBI Effect and E_2 is an Implicit TTD Memory Read Effect such that E_2 is in the Invalidation Scope of E_1, and the TLBI-after relation is asymmetric.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DC-before, DC-after

For two Effects E_1 and E_2, if all of the following applies:

- E_1 is a DC CVAU Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 and E_2 are to the Same Cache Line.

then one of the following applies:

- E_1 is *DC-after* E_2 (equivalently E_2 is *DC-before* E_1), or
- E_2 is *DC-after* E_1 (equivalently E_1 is *DC-before* E_2).

Note

The DC-after relation enumerates all possible pairs (E_1, E_2) where E_1 is a DC CVAU Effect and E_2 is an Explicit Memory Write Effect such that E_1 and E_2 are to the Same Cache Line, and the relation DC-after is asymmetric.

IC-before, IC-after

For two Effects E_1 and E_2, if and only if all of the following applies:

- E_1 is a completed IC Effect.
- E_2 is an Implicit Instruction Memory Read Effect.
- E_1 and E_2 are to the Same Cache Line.

then one and only one of the following applies:

- E_1 is *IC-after* E_2 (equivalently E_2 is *IC-before* E_1), or
- E_2 is *IC-after* E_1 (equivalently E_1 is *IC-before* E_2).

Note

The IC-after relation enumerates all possible pairs (E_1,E_2) where E_1 is a completed IC Effect and E_2 is an Implicit Instruction Memory Read Effect such that E_1 and E_2 are to the Same Cache Line, and the relation IC-after is asymmetric.

Single-copy-atomic grouping

A *single-copy-atomic grouping* gathers Explicit Memory Effects generated by the same instruction when the architecture requires the Explicit Memory Effects to be single-copy atomic as defined in Requirements for single-copy atomicity. For example, an aligned STRH instruction generates two Explicit Memory Write Effects that are in the same single-copy-atomic grouping.

Atomicity of Successful Read-Modify-Write pair

For two Effects E_1 and E_2 if all of the following apply:

- E_1 and E_2 form a successful Read-Modify-Write pair.
- There is an Explicit Memory Write Effect E_3.
- It is not the case that all of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 and E_3 are from the same Processing Element.
  - E_2 is an Explicit Memory Effect.
  - E_2 and E_3 are from the same Processing Element.

then it is not the case that E_1 is Coherence-before E_3 and E_3 is Coherence-before E_2.

B2.3.2 Intrinsic Dependency relations

Intrinsic Dependencies are as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Intrinsic Data Dependency

There is an *Intrinsic Data Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- It is not the case that E_1 is an Intrinsic Branching Effect.
- E_1 and E_2 are generated by the same instruction `I`.
- The instruction semantics for that instruction `I` indicate that the value produced by E_1 is consumed by E_2.

Note

This applies even in cases where the Intrinsic Data Dependency might be perceived to be a false dependency.

Intrinsic Control Dependency

There is an *Intrinsic Control Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- E_1 is an Intrinsic Branching Effect.
- E_1 and E_2 are generated by the same instruction `I`.
- The instruction semantics for that instruction `I` indicate that the generation of E_2 is conditional and based on the determination of E_1.

Intrinsic Order Dependency

There is an *Intrinsic Order Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- E_1 is a Memory Effect.
- E_1 and E_2 are generated by the same instruction `I`.
- The instruction semantics for that instruction `I` indicate that E_1 is generated before E_2.
- It is not the case that there is an Intrinsic Data Dependency from E_1 to E_2.
- It is not the case that there is an Intrinsic Control Dependency from E_1 to E_2.
- E_2 is a Memory Effect.

In the following, instructions whose Intrinsic dependencies are special cases that arose during the formalisation of the memory model are listed. This does not preclude the existence of other special cases in the future.

B2.3.2.1 Conditional Selection instructions

The following applies to the Effects generated by `CSEL` X_d,X_n,X_m, cond

When the condition `cond` is true all of the following apply:

- Dt_1: There is an Intrinsic Data Dependency from the Register Read Effect of `PSTATE.NZCV` to the Intrinsic Branching Effect corresponding to the condition `cond`.
- Ct_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks the condition `cond` to the Register Read Effect of X_n.
- Dt_2: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Register Write Effect of X_d.

When the condition `cond` is false all of the following apply:

- Df_1: There is an Intrinsic Data Dependency from the Register Read Effect of `PSTATE.NZCV` to the Intrinsic Branching Effect corresponding to the condition `cond`.
- Cf_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect corresponding to the condition `cond` to the Register Read Effect of X_m.
- Df_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_m to the Register Write Effect of X_d.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.2.2 Compare-and-Swap instructions

The following apply to the Effects generated by `CAS` X_s,X_t,[X_n]:

When the value of the Memory Location x addressed by X_n is not equal to the value of X_s one of the following applies:

- D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
- D_2: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
- D_3: There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
- Df_1: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_s.

When the value of the Memory Location x addressed by X_n is equal to the value of X_s one of the following applies:

- All of the following apply:
  - D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
  - D_2: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
  - D_3: There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
  - Ds_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Ds_2: There is an Intrinsic Data Dependency from Register Read Effect X_t to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Cs_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Ds_{31}: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_s.
- All of the following apply:
  - D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
  - D_2: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
  - D_3: There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
  - Ds_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Ds_2: There is an Intrinsic Data Dependency from the Register Read Effect of X_t to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Cs_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Ds_3: There is an Intrinsic Data Dependency from Register Read Effect X_s to the Register Write Effect of X_s.
  - Cs_2: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Register Write Effect of X_s.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.2.3 Swap instructions

All of the following apply to the Effects generated by the `SWP` X_s, X_t,[X_n] instruction:

- D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
- D_2: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
- O_1: There is an Intrinsic Order Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
- D_3: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_t.
- D_4: There is an Intrinsic Data Dependency from the Register Read Effect of X_s, to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.

B2.3.2.4 Branch with Link to Register instructions

All of the following apply to the Effects generated by the `BLR` X_n instruction:

- D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Branching Effect of the instruction.

B2.3.2.5 MTE Checked instructions

If FEAT_MTE2 is implemented, a Checked `LDR` X_t,[X_n] instruction has the following additional dependencies:

When Tag Check Faults are configured to raise a Synchronous Exception one of the following applies:

- If the Logical Address Tag is equal to the Allocation tag, all of the following apply:
  - — Dst_1: There is an Intrinsic Data Dependency from the Implicit Tag Memory Read Effect of the Tag Location x of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n.
  - — Cst_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
- If the Logical Address Tag is not equal to the Allocation tag, all of the following apply:
  - — Dsf_1: There is an Intrinsic Data Dependency from the Implicit Tag Memory Read Effect of the Tag Location x of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n.
  - — Csf_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n to the TagCheck Fault Effect.

When Tag Check Faults are configured to not raise a Synchronous Exception one of the following applies:

- If the Logical Address Tag is equal to the Allocation tag, all of the following apply:
  - — Dat_1: There is an Intrinsic Data Dependency from the Implicit Tag Memory Read Effect of the Tag Location x of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n.
  - — Cat_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n to the empty Effect.
- If the Logical Address Tag is not equal to the Allocation tag, all of the following apply:
  - — Daf_1: There is an Intrinsic Data Dependency from the Implicit Tag Memory Read Effect of the Tag Location x of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n.
  - — Caf_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n to the TagCheck Fault Effect.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.3 Tag-check-intrinsically-before

Tag-check-before

An Effect E_1 is *Tag-check-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Tag Memory Read Effect.
- There is an Intrinsic Data Dependency from E_1 to E_3.
- E_3 is a Branching Effect.
- There is an Intrinsic Control Dependency from E_3 to E_2.
- One of the following applies:
  - E_2 is an Explicit Memory Effect.
  - E_2 is a TagCheck Fault Effect.

Tag-check-intrinsically-before

An Effect E_1 is *Tag-check-intrinsically-before* an Effect E_2 if all of the following apply:

- E_1 is Tag-check-before E_2.
- It is not the case that E_2 is an Explicit Memory Read Effect.

B2.3.4 Translation-intrinsically-before

An Effect E_1 is *Translation-intrinsically-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit TTD Memory Read Effect.
- There is an Intrinsic Data Dependency from E_1 to E_3.
- E_3 is a Branching Effect.
- There is an Intrinsic Control Dependency from E_3 to E_2.
- One of the following applies:
  - E_2 is an Explicit Memory Effect.
  - E_2 is an MMU Fault Effect.

B2.3.5 Fetch-intrinsically-before

An Effect E_1 is *Fetch-intrinsically-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- There is an Intrinsic Data Dependency from E_1 to E_3.
- E_3 is a Branching Effect.
- One of the following applies:
  - There is an Intrinsic Control Dependency from E_3 to E_2.
  - All of the following apply:
    - There is an Intrinsic Control Dependency from E_3 to E_4.
    - There is an Intrinsic Data Dependency from E_4 to E_2.

B2.3.6 Dependency relations

Dependency through registers and memory

There is a *Dependency through registers and memory* from an Effect E_1 to an Effect E_2 if one of the following applies:

- All of the following apply:
  - It is not the case that E_1 is generated by a Store-Exclusive instruction as part of a successful Load-Exclusive/Store-Exclusive pair.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - E_2 is a Local register read successor of E_1.
- E_2 is a Local memory read successor of E_1.
- There is an Intrinsic Data Dependency from E_1 to E_2.
- All of the following apply:
  - There is a Dependency through registers and memory from E_1 to E_3.
  - There is a Dependency through registers and memory from E_3 to E_2.

Data dependency

There is a *Data dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2.
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the data value written by E_2.
- E_2 is an Explicit Memory Write Effect.

Address dependency

There is an *Address dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2.
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the address of the Location of E_2.
- One of the following applies:
  - E_2 is an Explicit Memory Effect.
  - E_2 is an Implicit Tag Memory Read Effect.
  - E_2 is an Implicit TTD Memory Read Effect.
  - E_2 is a Hardware Update Effect.
  - E_2 is a TLBI Effect.
  - E_2 is a DC CVAU Effect.
  - E_2 is an IC IVAU Effect.

Control dependency

There is a *Control dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_4.
- E_4 is a Conditional Branching Effect.
- E_4 appears in program order before E_2.

Pick dependency through registers and memory

There is a *Pick dependency through registers and memory* from an Effect E_1 to an Effect E_2 if one of the following applies:

- There is a Dependency through registers and memory from E_1 to E_2.
- There is an Intrinsic Control Dependency from E_1 to E_2.
- All of the following apply:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - There is a Pick dependency through registers and memory from E_1 to E_3.
    - There is a Pick dependency through registers and memory from E_3 to E_2.

Pick Basic dependency

There is a *Pick Basic dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- One of the following applies:
    - E_1 is an Explicit Memory Read Effect.
    - E_1 is a Register Read Effect.
- One of the following applies:
    - There is a Pick dependency through registers and memory from E_1 to E_2.
    - E_1 and E_2 are the same Effect.

Pick Data dependency

There is a *Pick Data dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- One of the following applies:
    - There is an Intrinsic Data Dependency from E_3 to E_2.
    - All of the following apply:
        - There is an Intrinsic Data Dependency from E_3 to E_4.
        - There is an Intrinsic Control Dependency from E_4 to E_2.
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the data value written by E_2.
- E_2 is an Explicit Memory Write Effect.

Pick Address dependency

There is a *Pick Address dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2.
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the address of the Location of E_2.
- One of the following applies:
    - E_2 is an Explicit Memory Effect.
    - E_2 is an Implicit Tag Memory Read Effect.
    - E_2 is an Implicit TTD Memory Read Effect.
    - E_2 is a Hardware Update Effect.
    - E_2 is a TLBI Effect.
    - E_2 is a DC CVAU Effect.
    - E_2 is an IC IVAU Effect.

Pick Control dependency

There is a *Pick Control dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_4 is a Conditional Branching Effect.
- E_4 appears in program order before E_2.

Pick dependency

There is a *Pick dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- One of the following applies:
  - There is a Pick Basic dependency from E_1 to E_2.
  - There is a Pick Address dependency from E_1 to E_2.
  - There is a Pick Data dependency from E_1 to E_2.
  - There is a Pick Control dependency from E_1 to E_2.
- It is not the case that E_1 and E_2 are generated by the same instruction.

B2.3.7 Ordering relations

Explicit-hazard-ordered-before

An Effect E_1 is *Explicit-hazard-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_1 appears in program order before E_3.
- E_1 and E_3 are to the Same Location.
- E_3 is an Explicit Memory Read Effect.
- One of the following applies:
  - E_3 belongs to the same single-copy-atomic grouping as E_4.
  - E_3 and E_4 are the same Effect.
- E_4 is an Explicit Memory Read Effect.
- E_4 is Coherence-before E_5.
- E_4 and E_5 are from different Processing Elements.
- E_5 is an Explicit Memory Write Effect.
- One of the following applies:
  - E_5 belongs to the same single-copy-atomic grouping as E_2.
  - E_5 and E_2 are the same Effect.
- E_2 is an Explicit Memory Write Effect.

Note

The Explicit-hazard-ordered-before relation does not apply when either E_1 or E_2 or both are generated by SVE instructions.

TTD-read-ordered-before

An Effect E_1 is *TTD-read-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is TLBI-before E_3.
  - E_3 is a TLBI Effect.
  - E_3 appears in program order before E_4.
  - E_4 is a DSB FULL Effect.
  - E_4 appears in program order before E_2.
  - It is not the case that E_2 is an Implicit Memory Effect.
- All of the following apply:
  - E_1 is TLBI-before E_3.
  - E_3 is a TLBI Effect.
  - E_3 appears in program order before E_4.
  - E_4 is a DSB FULL Effect.
  - E_4 appears in program order before E_5.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - E_5 is an Instruction Fetch Barrier Effect.
  - E_5 appears in program order before E_2.
  - E_2 is an Implicit Memory Effect.
- All of the following apply:
  - FEAT_ETS2 is implemented.
  - E_1 is TLBI-before E_3.
  - E_3 is a TLBI Effect.
  - E_3 appears in program order before E_4.
  - E_4 is a DSB FULL Effect.
  - E_4 appears in program order before E_2.
  - E_2 is an Implicit TTD Memory Effect.

Same Virtual Address

An Effect E_1 and an Effect E_2 are to the *Same Virtual Address* if all of the following apply:

- E_1 is an Implicit TTD Memory Read Effect.
- E_1 is Translation-intrinsically-before E_3.
- E_3 and E_4 have the Same Low Order Bits.
- E_2 is Translation-intrinsically-before E_4.
- E_2 is an Implicit TTD Memory Read Effect.
- E_1 and E_2 are to the same Physical Address.

TLBI-ordered-before

An Effect E_1 is *TLBI-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is TTD-read-ordered-before E_2.
- All of the following apply:
  - E_3 is Translation-intrinsically-before E_1.
  - E_3 is TTD-read-ordered-before E_2.
  - E_3 and E_2 are from different Processing Elements.
- All of the following apply:
  - E_1 appears in program order before E_3 and E_1 and E_3 are to the Same Virtual Address.
  - E_3 is TTD-read-ordered-before E_2.
  - E_3 and E_2 are from different Processing Elements.

Instruction-read-ordered-before

An Effect E_1 is *Instruction-read-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is IC-before E_3.
  - E_3 is an IC Effect.
  - E_3 appears in program order before E_4.
  - E_4 is a DSB FULL Effect.
  - E_4 appears in program order before E_2.
  - It is not the case that E_2 is an Implicit Memory Effect.
- All of the following apply:
  - DIC is implemented.
  - E_1 is Coherence-before E_2.

IC-ordered-before

An Effect E_1 is *IC-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- E_1 appears in program order before E_3.
- E_3 is an Implicit Instruction Memory Read Effect.
- E_3 is Instruction-read-ordered-before E_2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Hazard-ordered-before

An Effect E_1 is *Hazard-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Explicit-hazard-ordered-before E_2.
- E_1 is TLBI-ordered-before E_2.
- E_1 is IC-ordered-before E_2.

ETS-ordered-before

An Effect E_1 is *ETS-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - FEAT_ETS2 is implemented.
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a TLBUncacheable Fault Effect.
  - E_2 is Translation-intrinsically-before E_3.
  - E_2 is an Implicit TTD Memory Read Effect.
- All of the following apply:
  - FEAT_ETS3 is implemented.
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_3.
  - E_3 is an MMU Fault Effect.
  - E_2 is Translation-intrinsically-before E_3.
  - E_2 is an Implicit TTD Memory Read Effect.
- All of the following apply:
  - FEAT_ETS3 is implemented.
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a TagCheck Fault Effect that generates a synchronous exception.
  - E_2 is Tag-check-intrinsically-before E_3.
  - E_2 is an Implicit Tag Memory Read Effect.

Note

The ETS-ordered-before relation does not apply when E_1 is generated by an Advanced SIMD, floating-point, SVE, or SME instruction. It also does not apply when E_2 is an Implicit TTD Memory Read Effect generated for an instruction fetch.

Fetch-ordered-before

An Effect E_1 is *Fetch-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- E_1 appears in program order before E_2.
- It is not the case that E_2 is an Implicit Instruction Memory Read Effect.

Same-Cache-Line-ordered-before

An Effect E_1 is *Same-Cache-Line-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_2.
  - E_1 and E_2 are to the Same Cache Line.
  - E_2 is a DC CVAU Effect.
- All of the following apply:
  - E_1 is a DC CVAU Effect.
  - E_1 appears in program order before E_2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - E_1 and E_2 are to the Same Cache Line.
    - E_2 is an Explicit Memory Effect.
- All of the following apply:
    - E_1 is a DC CVAU Effect.
    - E_1 appears in program order before E_2.
    - E_1 and E_2 are to the Same Cache Line.
    - E_2 is a DC CVAU Effect.

DSB-ordered-before

An Effect E_1 is *DSB-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
    - One of the following applies:
        - E_1 is a Memory Effect.
        - E_1 is a DC CVAU Effect.
        - E_1 is an IC Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DSB FULL Effect.
    - E_3 appears in program order before E_2.
    - It is not the case that one of the following applies:
        - E_2 is an Implicit TTD Memory Effect.
        - E_2 is an Implicit Instruction Memory Read Effect.
- All of the following apply:
    - FEAT_ETS2 is implemented.
    - One of the following applies:
        - E_1 is a Memory Effect.
        - E_1 is a DC CVAU Effect.
        - E_1 is an IC Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DSB FULL Effect.
    - E_3 appears in program order before E_2.
    - E_2 is an Implicit TTD Memory Effect.
- All of the following apply:
    - One of the following applies:
        - All of the following apply:
            - E_1 is an Explicit Memory Read Effect.
            - It is not the case that E_1 is generated by an instruction whose destination register is WZR or XZR.
        - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DSB LD Effect.
    - E_3 appears in program order before E_2.
    - It is not the case that one of the following applies:
        - E_2 is an Implicit TTD Memory Effect.
        - E_2 is an Implicit Instruction Memory Read Effect.
- All of the following apply:
    - FEAT_ETS2 is implemented.
    - E_1 is an Explicit Memory Read Effect.
    - It is not the case that E_1 is generated by an instruction whose destination register is WZR or XZR.
    - E_1 appears in program order before E_3.
    - E_3 is a DSB LD Effect.
    - E_3 appears in program order before E_2.
    - E_2 is an Implicit TTD Memory Effect.
- All of the following apply:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - E_1 is an Explicit Memory Write Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DSB ST Effect.
    - E_3 appears in program order before E_2.
    - It is not the case that one of the following applies:
        - E_2 is an Implicit TTD Memory Effect.
        - E_2 is an Implicit Instruction Memory Read Effect.
- All of the following apply:
    - FEAT_ETS2 is implemented.
    - E_1 is an Explicit Memory Write Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DSB ST Effect.
    - E_3 appears in program order before E_2.
    - E_2 is an Implicit TTD Memory Effect.

Instruction-fetch-barrier-ordered-before

An Effect E_1 is *Instruction-fetch-barrier-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
    - E_1 is an Explicit Memory Read Effect.
    - There is a Control dependency from E_1 to E_3.
    - E_3 is an Instruction Fetch Barrier Effect.
    - E_3 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Explicit Memory Read Effect.
    - There is a Pick Control dependency from E_1 to E_3.
    - E_3 is an Instruction Fetch Barrier Effect.
    - E_3 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Explicit Memory Read Effect.
    - There is an Address dependency from E_1 to E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_4.
    - E_4 is an Instruction Fetch Barrier Effect.
    - E_4 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Explicit Memory Read Effect.
    - There is a Pick Address dependency from E_1 to E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_4.
    - E_4 is an Instruction Fetch Barrier Effect.
    - E_4 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Explicit Memory Read Effect.
    - There is a Pick Address dependency from E_1 to E_3.
    - One of the following applies:
        - E_3 is Tag-check-intrinsically-before E_4.
        - E_3 is Translation-intrinsically-before E_4.
    - E_4 is an Instruction Fetch Barrier Effect.
    - E_4 appears in program order before E_2.
- All of the following apply:
    - E_1 is DSB-ordered-before E_3.
    - E_3 is an Instruction Fetch Barrier Effect.
    - E_3 appears in program order before E_2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All of the following apply:
  - E_1 is an Implicit TTD Memory Read Effect.
  - E_1 is Translation-intrinsically-before E_3.
  - E_3 is an Instruction Fetch Barrier Effect.
  - E_3 appears in program order before E_2.
- All of the following apply:
  - E_1 is an Implicit Tag Memory Read Effect.
  - E_1 is Tag-check-intrinsically-before E_3.
  - E_3 is an Instruction Fetch Barrier Effect.
  - E_3 appears in program order before E_2.
- All of the following apply:
  - E_1 is an Implicit TTD Memory Read Effect.
  - E_1 is Translation-intrinsically-before E_3.
  - E_3 is an Explicit Memory Effect.
  - E_3 appears in program order before E_4.
  - E_4 is an Instruction Fetch Barrier Effect.
  - E_4 appears in program order before E_2.
- All of the following apply:
  - E_1 is an Implicit Tag Memory Read Effect.
  - E_1 is Tag-check-before E_3.
  - E_3 is an Explicit Memory Effect.
  - E_3 appears in program order before E_4.
  - E_4 is an Instruction Fetch Barrier Effect.
  - E_4 appears in program order before E_2.

Dependency-ordered-before

An Effect E_1 is *Dependency-ordered-before* an Effect E_2 if one of the following applies:

- There is an Address dependency from E_1 to E_2.
- There is a Data dependency from E_1 to E_2.
- All of the following apply:
  - There is a Control dependency from E_1 to E_2.
  - One of the following applies:
    - E_2 is an Explicit Memory Write Effect.
    - E_2 is a Hardware Update Effect.
    - E_2 is a TLBI Effect.
    - E_2 is a DC CVAU Effect.
    - E_2 is an IC Effect.
- All of the following apply:
  - There is an Address dependency from E_1 to E_3.
  - E_3 is an Explicit Memory Effect.
  - E_3 appears in program order before E_2.
  - One of the following applies:
    - E_2 is an Explicit Memory Write Effect.
    - E_2 is a Hardware Update Effect.
- All of the following apply:
  - There is an Address dependency from E_1 to E_3.
  - E_3 is an Explicit Memory Effect.
  - E_2 is a Local memory read successor of E_3.
  - One of the following applies:
    - E_2 is an Explicit Memory Read Effect.
    - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
  - There is a Data dependency from E_1 to E_3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - E_3 is an Explicit Memory Effect.
    - E_2 is a Local memory read successor of E_3.
    - One of the following applies:
        - E_2 is an Explicit Memory Read Effect.
        - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
    - E_1 is an Implicit TTD Memory Read Effect.
    - E_1 is Translation-intrinsically-before E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.
- All of the following apply:
    - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 is Tag-check-before E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.

Pick-ordered-before

An Effect E_1 is *Pick-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
    - There is a Pick Address dependency from E_1 to E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.
        - E_2 is a TLBI Effect.
        - E_2 is a DC CVAU Effect.
        - E_2 is an IC Effect.
- There is a Pick Data dependency from E_1 to E_2.
- All of the following apply:
    - There is a Pick Control dependency from E_1 to E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.
        - E_2 is a TLBI Effect.
        - E_2 is a DC CVAU Effect.
        - E_2 is an IC Effect.
- All of the following apply:
    - There is a Pick Address dependency from E_1 to E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.

Atomic-ordered-before

E_1 and E_2 form a successful Read-Modify-Write pair if and only if E_1 and E_2 are generated by the same atomic instruction or the same successful Load-Exclusive/Store-Exclusive instruction pair to the Same Location.

An Effect E_1 is *Atomic-ordered-before* an Effect E_2 if one of the following applies:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 and E_2 form a successful Read-Modify-Write pair.
  - E_2 is an Explicit Memory Effect.
- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 and E_3 form a successful Read-Modify-Write pair.
  - E_2 is a Local memory read successor of E_3.
  - One of the following applies:
    - E_2 is an Explicit Memory Read Effect with Acquire semantics.
    - E_2 is an Explicit Memory Read Effect with AcquirePC semantics.
- All of the following apply:
  - E_1 is an Implicit TTD Memory Read Effect.
  - E_1 and E_2 form a successful Read-Modify-Write pair.
  - E_2 is a Hardware Update Effect.

Barrier-ordered-before

An Effect E_1 is *Barrier-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - One of the following applies:
    - E_1 is an Explicit Memory Effect.
    - E_1 is an Implicit Tag Memory Read Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DMB FULL Effect.
  - E_3 appears in program order before E_2.
  - One of the following applies:
    - E_2 is an Explicit Memory Effect.
    - E_2 is an Implicit Tag Memory Read Effect.
    - E_2 is an MMU Fault Effect.
- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DMB FULL Effect.
  - E_3 appears in program order before E_2.
  - E_2 is a DC CVAU Effect.
- All of the following apply:
  - E_1 is a DC CVAU Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DMB FULL Effect.
  - E_3 appears in program order before E_2.
  - E_2 is an Explicit Memory Effect.
- All of the following apply:
  - E_1 is a DC CVAU Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DMB FULL Effect.
  - E_3 appears in program order before E_2.
  - E_2 is a DC CVAU Effect.
- All of the following apply:
  - One of the following applies:
    - All of the following apply:
      - E_1 is an Explicit Effect.
      - E_1 is a Memory Read Effect.
      - It is not the case that E_1 is generated by an instruction whose destination register is WZR or XZR.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

        - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DMB LD Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Effect.
        - E_2 is an Implicit Tag Memory Read Effect.
        - E_2 is an MMU Fault Effect.
- All of the following apply:
    - E_1 is an Explicit Memory Write Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DMB ST Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is an MMU Fault Effect.
- All of the following apply:
    - One of the following applies:
        - E_1 is an Explicit Memory Write Effect with Release semantics.
        - E_1 is a Fault Effect with Release semantics.
    - E_3 and E_1 are generated by an atomic instruction.
    - E_3 is an Explicit Memory Read Effect with Acquire semantics.
    - E_1 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Effect.
        - E_2 is an Implicit Tag Memory Read Effect.
        - E_2 is an MMU Fault Effect.
- All of the following apply:
    - One of the following applies:
        - E_1 is an Explicit Memory Write Effect with Release semantics.
        - E_1 is a Fault Effect with Release semantics.
    - E_1 appears in program order before E_2.
    - E_2 is an Explicit Memory Read Effect with Acquire semantics.
- All of the following apply:
    - One of the following applies:
        - E_1 is an Explicit Memory Read Effect with Acquire semantics.
        - E_1 is an Explicit Memory Read Effect with AcquirePC semantics.
    - E_1 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Effect.
        - E_2 is an Implicit Tag Memory Read Effect.
        - E_2 is an MMU Fault Effect.
- All of the following apply:
    - One of the following applies:
        - E_1 is an Explicit Memory Read Effect with Acquire semantics.
        - E_1 is an Explicit Memory Read Effect with AcquirePC semantics.
    - There is an Intrinsic Order Dependency from E_1 to E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Effect.
        - E_2 is an Implicit Tag Memory Read Effect.
        - E_2 is an MMU Fault Effect.
- All of the following apply:
    - One of the following applies:
        - E_1 is an Explicit Memory Effect.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

        - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect with Release semantics.
        - E_2 is a Fault Effect with Release semantics.
- All of the following apply:
    - One of the following applies:
        - E_1 is an Explicit Memory Effect.
        - E_1 is an Implicit Tag Memory Read Effect.
    - There is an Intrinsic Order Dependency from E_1 to E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect with Release semantics.
        - E_2 is a Fault Effect with Release semantics.

Locally-ordered-before

An Effect E_1 is *Locally-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Tag-check-intrinsically-before E_2.
- E_1 is Translation-intrinsically-before E_2.
- E_1 is Fetch-intrinsically-before E_2.
- E_1 is ETS-ordered-before E_2.
- E_1 is Fetch-ordered-before E_2.
- E_1 is Same-Cache-Line-ordered-before E_2.
- E_1 is DSB-ordered-before E_2.
- E_1 is Instruction-fetch-barrier-ordered-before E_2.
- E_2 is a Local memory write successor of E_1.
- All of the following apply:
    - E_3 is a Local memory write successor of E_1.
    - E_3 belongs to the same single-copy-atomic grouping as E_2.
- E_2 is a Local MMU Fault successor of E_1.
- E_1 is Dependency-ordered-before E_2.
- E_1 is Pick-ordered-before E_2.
- E_1 is Atomic-ordered-before E_2.
- E_1 is Barrier-ordered-before E_2.
- All of the following apply:
    - E_1 is Locally-ordered-before E_3.
    - E_3 is Locally-ordered-before E_2.

Pick-locally-ordered-before

An Effect E_1 is *Pick-locally-ordered-before* an Effect E_2 if all of the following apply:

- There is a Pick dependency from E_1 to E_3.
- E_3 is Locally-ordered-before E_2.
- E_2 is an Explicit Memory Write Effect.

Locally-hardware-required-ordered-before

An Effect E_1 is *Locally-hardware-required-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Locally-ordered-before E_2.
- E_1 is Pick-locally-ordered-before E_2.
- All of the following apply:
    - E_1 is Locally-hardware-required-ordered-before E_3.
    - E_3 is Locally-hardware-required-ordered-before E_2.

Hardware-required-ordered-before

An Effect E_1 is *Hardware-required-ordered-before* an Effect E_2 if one of the following applies:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_1 is Locally-hardware-required-ordered-before E_2.
- E_1 is Hazard-ordered-before E_2.

B2.3.8 Observation relations

Explicit-Observed-by

An Effect E_1 is *Explicit-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_2 Reads-from-memory E_1.
  - E_1 and E_2 are from different Processing Elements.
  - E_2 is an Explicit Memory Effect.
- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 is Coherence-before E_2.
  - E_1 and E_2 are from different Processing Elements.
  - E_2 is an Explicit Memory Effect.

Tag-Observed-by

An Effect E_1 is *Tag-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is an Explicit Memory Write Effect.
  - E_2 Reads-from-memory E_1.
  - E_1 and E_2 are from different Processing Elements.
  - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
  - E_1 is an Implicit Tag Memory Read Effect.
  - E_1 is Coherence-before E_2.
  - E_1 and E_2 are from different Processing Elements.
  - E_2 is an Explicit Memory Write Effect.

TLBuncacheable-coherence-before

An Effect E_1 is *TLBUncacheable-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit TTD Memory Read Effect.
- E_1 is Translation-intrinsically-before E_3.
- E_3 is a TLBUncacheable Fault Effect.
- E_1 is Coherence-before E_2.
- One of the following applies:
  - E_2 is an Explicit Memory Write Effect.
  - E_2 is a Hardware Update Effect.

Hardware-Update-coherence-before

An Effect E_1 is *Hardware-Update-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_1 is Coherence-before E_2.
- E_2 is a Hardware Update Effect.

TLBI-coherence-before

An Effect E_1 is *TLBI-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is a TLBI Effect.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_1 is TLBI-before E_3.
- E_3 is an Implicit TTD Memory Read Effect.
- E_3 is Coherence-before E_2.
- E_2 is a Memory Write Effect.

TTD-Observed-by

An Effect E_1 is *TTD-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is an Implicit TTD Memory Effect.
  - E_2 Reads-from-memory E_1.
- All of the following apply:
  - E_2 Reads-from-memory E_1.
  - E_2 is an Implicit TTD Memory Effect.
- E_1 is TLBUncacheable-coherence-before E_2.
- E_1 is Hardware-Update-coherence-before E_2.
- All of the following apply:
  - E_1 is a Hardware Update Effect.
  - E_1 is Coherence-before E_2.
  - E_2 is a Memory Write Effect.
- All of the following apply:
  - E_1 is a Memory Write Effect.
  - E_1 is Coherence-before E_2.
  - E_2 is a Hardware Update Effect.
- E_1 is TLBI-coherence-before E_2.

IC-Coherence-before

An Effect E_1 is *IC-coherence-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - it is not the case that DIC is implemented and it is not the case that IDC is implemented.
  - E_1 is an IC Effect.
  - E_1 is IC-before E_3.
  - E_3 is an Implicit Instruction Memory Read Effect.
  - E_3 is Coherence-before E_4.
  - E_4 is a Memory Write Effect.
  - E_4 is DC-before E_2.
  - E_2 is a DC CVAU Effect.
- All of the following apply:
  - it is not the case that DIC is implemented and IDC is implemented.
  - E_1 is an IC Effect.
  - E_1 is IC-before E_3.
  - E_3 is an Implicit Instruction Memory Read Effect.
  - E_3 is Coherence-before E_2.
  - E_2 is a Memory Write Effect.
- All of the following apply:
  - DIC is implemented and IDC is implemented.
  - E_1 is an Implicit Instruction Memory Read Effect.
  - E_1 is Coherence-before E_2.
  - E_2 is a Memory Write Effect.

Instruction-Observed-by

An Effect E_1 is *Instruction-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - E_2 Reads-from-memory E_1.
  - E_2 is an Implicit Instruction Memory Read Effect.
- E_1 is IC-before E_2.
- All of the following apply:
  - E_1 is a DC CVAU Effect.
  - E_1 is DC-before E_2.
  - E_2 is a Memory Write Effect.
- All of the following apply:
  - E_1 is a Memory Write Effect.
  - E_1 is DC-before E_2.
  - E_2 is a DC CVAU Effect.
- E_1 is IC-coherence-before E_2.

Observed-by

An Effect E_1 is *Observed-by* an Effect E_2 if one of the following applies:

- E_1 is Explicit-Observed-by either E_2 or an Effect which belongs to the same single-copy-atomic grouping as E_2.
- E_1 is Tag-Observed-by either E_2 or an Effect which belongs to the same single-copy-atomic grouping as E_2.
- E_1 is TTD-Observed-by E_2.
- E_1 is Instruction-Observed-by E_2.

B2.3.9 Definition of an Architecturally Allowed Execution

An Architecturally Allowed Execution must satisfy the *External visibility requirement*, which is formulated in terms of an ordering relation called *Ordered-before*. An execution that does not satisfy the external visibility requirement is an Architecturally Forbidden Execution.

Ordered-before

An Effect E_1 is *Ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Hardware-required-ordered-before E_2.
- E_1 is Observed-by E_2.
- All of the following apply:
  - E_1 is Ordered-before E_3.
  - E_3 is Ordered-before E_2.

External visibility requirement

The external visibility requirement is defined as follows: an Architecturally Allowed Execution must not exhibit a cycle in the Ordered-before relation.

In particular, for an Effect E_1 that is Ordered-before an Effect E_2, the External visibility requirement requires that E_2 is not Observed-by E_1.

B2.3.10 Alternative formulations of the External visibility requirement for Explicit Memory Effects

B2.3.10.1 Additional ordering relations

Single-copy-atomic-ordered-before

A Memory Read Effect E_1 is *single-copy-atomic-ordered-before* another Memory Read Effect E_2 if all the following apply:

- E_1 belongs to the same single-copy-atomic grouping as E_2.
- E_1 Reads-from-memory a Memory Write Effect E_3.
- E_1 and E_3 are from different Processing Elements.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_2 Reads-from-memory a Memory Write Effect E_4.
- E_2 and E_4 are from the same Processing Element.

B2.3.10.2 External completion requirement for Explicit Memory Effects

The *External completion requirement for Explicit Memory Effects* is an alternative way of formulating the External visibility requirement, in the context of Explicit Memory Effects only.

The Completes-before order is a total order that corresponds to the order in which Memory Effects complete within the system. Effects E_1 and E_2 constitute a single entry in the Completes-before order, if any of the following applies:

- All of the following apply:
  - E_1 is a Memory Write Effect.
  - E_2 is a Memory Write Effect.
  - E_1 belongs to the same single-copy-atomic grouping as E_2.
- All of the following apply:
  - E_1 is a Memory Read Effect.
  - E_2 is a Memory Read Effect.
  - E_1 belongs to the same single-copy-atomic grouping as E_2.
  - E_1 Reads-from-memory a Memory Write Effect E_3.
  - E_1 and E_3 are from different Processing Elements.
  - E_2 Reads-from-memory a Memory Write Effect E_4.
  - E_2 and E_4 are from different Processing Elements.
- All of the following apply:
  - E_1 is a Memory Read Effect.
  - E_2 is a Memory Read Effect.
  - E_1 belongs to the same single-copy-atomic grouping as E_2.
  - E_1 Reads-from-memory a Memory Write Effect E_3.
  - E_1 and E_3 are from the same Processing Element.
  - E_2 Reads-from-memory a Memory Write Effect E_4.
  - E_2 and E_4 are from the same Processing Element.
  - E_3 belongs to the same single-copy-atomic grouping as E_4.

All other Memory Read Effects constitute distinct entries in the Completes-before order.

Completes-before

A Memory Read or Write Effect E_1 completes-before a Memory Read or Write Effect E_2 if E_1 appears in the *Completes-before* order before E_2.

Deriving Reads-from-memory and Coherence order from the Completes-before order

The Reads-from-memory relation can be derived from the Completes-before order by specifying which Memory Write Effect every Memory Read Effect gets its value from. Specifically, for a Memory Read Effect E_1 one of the following must apply:

- If all of the following apply:
  - There is a Memory Write Effect E_2.
  - E_1 is a Local memory read successor of E_2.
  - E_1 Completes-before E_2.
  - There is no Memory Write Effect E_3 to the Same Location in the Completes-before order between E_2 and an Explicit Memory Read Effect E_4 to the Same Location that appears in program order before E_1.

  then it must be that E_1 Reads-from-memory E_2.
- If all of the following apply:
  - There is a Memory Write Effect E_2 (possibly representing the writing of the initial value to the Location).
  - E_2 and E_1 are to the Same Location.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— E_2 Completes-before E_1.
— It is not the case that E_2 Completes-before a Memory Write Effect E_3 and E_1 is the Local memory read successor of E_3.
— There is no Memory Write Effect E_4 to the Same Location in the Completes-before order between E_2 and E_1.
— There is no Memory Write Effect E_5 to the Same Location in the Completes-before order between E_2 and a Memory Read Effect E_6 to the Same Location that appears in program order before E_1.

then it must be that E_1 Reads-from-memory E_2.

The Coherence order can be derived from the Completes-before order by letting the Coherence order of Memory Write Effects to a memory location be the order in which those Memory Write Effects appear in the Completes-before order. The final value of each memory location is therefore determined by the final Memory Write Effect to each Location in the Completes-before order. If no such Memory Write Effect exists for a given Location, the final value is the initial value of that Location.

External completion requirement for Explicit Memory Effects

The External completion requirement for Explicit Memory Effects requires that an Memory Effect E_1 Completes-before a Memory Effect E_2 if any of the following statements are true:

- E_1 is Locally-hardware-required-ordered-before E_2.
- E_1 is a Memory Read Effect, E_2 is a Memory Read Effect, and E_1 is single-copy-atomic-ordered-before E_2.

B2.3.10.3 External global completion requirement for Explicit Memory Effects

The *External global completion requirement for Explicit Memory Effects* is an alternative way of formulating the External visibility requirement, in the context of Explicit Memory Effects only.

The Globally-completes-before order is a total order that corresponds to the order in which Memory Effects complete within the system. Effects E_1 and E_2 constitute a single entry in the Globally-completes-before order, if any of the following applies:

- All of the following apply:
  — E_1 is a Memory Write Effect.
  — E_2 is a Memory Write Effect.
  — E_1 belongs to the same single-copy-atomic grouping as E_2.
- All of the following apply:
  — E_1 is a Memory Read Effect.
  — E_2 is a Memory Read Effect.
  — E_1 belongs to the same single-copy-atomic grouping as E_2.
  — E_1 Reads-from-memory a Memory Write Effect E_3.
  — E_1 and E_3 are from different Processing Elements.
  — E_2 Reads-from-memory a Memory Write Effect E_4.
  — E_2 and E_4 are from different Processing Elements.
- All of the following apply:
  — E_1 is a Memory Read Effect.
  — E_2 is a Memory Read Effect.
  — E_1 belongs to the same single-copy-atomic grouping as E_2.
  — E_1 Reads-from-memory a Memory Write Effect E_3.
  — E_1 and E_3 are from the same Processing Element.
  — E_2 Reads-from-memory a Memory Write Effect E_4.
  — E_2 and E_4 are from the same Processing Element.
  — E_3 belongs to the same single-copy-atomic grouping as E_4.

All other Memory Read Effects constitute distinct entries in the Globally-completes-before order.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Globally-completes-before

A Memory Read or Write Effect E_1 *Globally-completes-before* a Memory Read or Write Effect E_2 if E_1 appears in the Globally-completes-before order before E_2.

Deriving Reads-from-memory and Coherence order from the Globally-completes-before order

The Reads-from-memory relation can be derived from the Globally-completes-before order by specifying which Memory Write Effect every Memory Read Effect gets its value from. Specifically, for a Memory Read Effect E_1 it must be that if all of the following applies:

- There is Memory Write Effect E_2 (possibly representing the writing of the initial value to the Location).
- E_2 Globally-completes-before E_1.
- E_1 and E_2 are to the Same Location.
- There is no Memory Write Effect E_3 to the Same Location in the Globally-completes-before order between E_2 and E_1.
- There is no Memory Write Effect E_4 to the Same Location in the Globally-completes-before order between E_2 and an Explicit Memory Read Effect E_5 to the Same Location that appears in program order before E_1.

then it must be that E_1 Reads-from-memory E_2.

The Coherence order can be derived from the Globally-completes-before order by letting the Coherence order of Memory Write Effects to a memory location be the order in which those Memory Write Effects appear in the Globally-completes-before order. The final value of each memory location is therefore determined by the final Memory Write Effect to each Location in the Globally-completes-before order. If no such Memory Write Effect exists for a given Location, the final value is the initial value of that Location.

External global completion requirement

The External global completion requirement requires that a Memory Effect E_1 Globally-completes-before a Memory Effect E_2 if any of the following statements are true:

- E_1 is Locally-hardware-required-ordered-before E_2 and one of the following applies:
  - E_1 is a Memory Write Effect.
  - E_1 is a Memory Read Effect, and one of the following applies:
    - E_1 Reads-from-memory a Memory Write Effect E_3, and E_1 and E_3 are from different Processing Elements, or.
    - E_1 Reads-from-memory a Memory Write Effect E_3, E_1, and E_3 are from the same Processing Element, and E_3 is Locally-hardware-required-ordered-before E_2.
- E_1 is a Memory Read Effect, E_2 is a Memory Read Effect, and E_1 is single-copy-atomic-ordered before E_2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.4 Additional ordering requirements outside of the scope of the formal concurrency model

The following sections present ordering rules which have been architected but not yet formalized. As a consequence they do not appear in the formal concurrency model, nor in the corresponding transliteration in Ordering requirements defined by the formal concurrency model. They will be integrated into Ordering requirements defined by the formal concurrency model in due course.

B2.4.1 Completion and endpoint ordering

For all memory, the completion rules are defined as:

- A Memory Read effect E_1 to a Location is complete for a shareability domain when all of the following are true:
  - Any Memory Write Effect to the same Location by an observer within the shareability domain will be Coherence-after E_1.
  - All Implicit TTD Memory Read Effects associated with E_1 are complete for that shareability domain.
- A Memory Write effect E_1 to a Location is complete for a shareability domain when all of the following are true:
  - Any Memory Write Effect to the same Location by an observer within the shareability domain will be Coherence-after E_1.
  - Any Memory Read Effect to the same Location by an observer within the shareability domain will be such that either Reads-from E_1 or Reads-from a Memory Write effect that is Coherence-after E_1.
  - All Implicit TTD Memory Read Effects associated with E_1 are complete for that shareability domain.
- An Implicit TTD Memory Read Effect is complete for a shareability domain when the memory accesses associated with the Implicit TTD Memory Read Effect, including Hardware Update Effects, are complete for that shareability domain, and the TLB is updated.
- A cache maintenance instruction is complete for a shareability domain when the memory effects of the instruction are complete for that shareability domain, and Implicit TTD Memory Read Effects that arise from the instruction are complete for that shareability domain.
- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated are complete.

The completion of any cache or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction.

Additionally, for Device-nGnRnE memory, a read or write of a Location in a Memory-mapped peripheral that exhibits side effects is complete only when the read or write both:

- Can begin to affect the state of the Memory-mapped peripheral.
- Can trigger all associated side effects, whether they affect other peripheral devices, PEs, or memory.

Interaction between observers in a system is not restricted to communication via shared variables in coherent memory. For example, an observer could configure an interrupt controller to raise an interrupt on another observer as a form of message passing. These interactions typically involve an additional agent, which defines the instruction sequence that is required to establish communication links between different observers. When these forms of interaction are used in conjunction with shared variables, a `DSB` instruction can be used to enforce ordering between them.

Note

This requirement for Device-nGnRnE memory is consistent with the memory access having reached the peripheral endpoint.

Note

These completion rules mean that, for example, a cache maintenance instruction that operates by VA to the PoC completes only after memory at the PoC has been updated.

B2.4.1.1 Peripherals

This section defines a Memory-mapped peripheral and the total order of reads and writes to a peripheral which is defined as the Peripheral arrival order:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Memory-mapped peripheral

A *Memory-mapped peripheral* occupies a memory region of IMPLEMENTATION DEFINED size and can be accessed using load and store instructions. Memory effects to a Memory-mapped peripheral can have side effects, such as causing the peripheral to perform an action. Values that are read from addresses within a Memory-mapped peripheral might not correspond to the last data value written to those addresses. As such, Memory effects to a Memory-mapped peripheral might not appear in the Reads-from or Coherence order relations.

Peripheral arrival order

The *Peripheral arrival order* of a Memory-mapped peripheral is a total order on all Memory Read Effects and Memory Write Effects to that peripheral.

For a Memory Effect E_1 and a Memory Effect E_2 to the same peripheral if any of the following applies:

- All of the following apply:
  - Any of the following applies:
    - E_1 is to Device-nGnRE Memory.
    - E_1 is to Device-nGnRnE Memory.
  - Any of the following applies:
    - E_2 is to Device-nGnRE Memory.
    - E_2 is to Device-nGnRnE Memory.
  - Any of the following applies:
    - FEAT_XS is not implemented.
    - All of the following apply:
      - FEAT_XS is implemented.
      - E_1 has the XS attribute.
      - E_2 has the XS attribute.
    - All of the following apply:
      - FEAT_XS is implemented.
      - E_1 does not have the XS attribute.
      - E_2 does not have the XS attribute.
  - E_1 is in program-order before E_2.
- All of the following apply:
  - Any of the following applies:
    - E_1 is to Device-nGRE Memory.
    - E_1 is to Device-nGnRE Memory.
    - E_1 is to Device-nGnRnE Memory.
  - Any of the following applies:
    - E_2 is to Device-nGRE Memory.
    - E_2 is to Device-nGnRE Memory.
    - E_2 is to Device-nGnRnE Memory.
  - E_1 is Ordered-before E_2.
- All of the following apply:
  - Any of the following applies:
    - E_1 is to Device Memory.
    - E_1 is to Non-cacheable Normal Memory.
  - Any of the following applies:
    - E_2 is to Device Memory.
    - E_2 is to Non-cacheable Normal Memory.
  - E_1 is Barrier-ordered-before E_2.

Then E_1 is inserted in the Peripheral arrival order for the peripheral before E_2 and it is not permitted that E_1 and E_2 are gathered into a single transaction. Otherwise, exactly one of following applies:

- E_1 and E_2 are gathered into a single transaction and appear as one Effect in Peripheral arrival order for the peripheral.
- E_1 is inserted in the Peripheral arrival order for the peripheral before E_2.
- E_2 is inserted in the Peripheral arrival order for the peripheral before E_1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

When FEAT_XS is implemented, if E_1 and E_2 are to Device-nGnRE Memory or to Device-nGnRnE Memory and E_1 and E_2 are within the same Memory-mapped peripheral, but E_1 and E_2 use different XS attribute, the order of arrival at the endpoint is not defined by the architecture.

Out-of-band-ordered-before

An Effect E_1 is *Out-of-band-ordered-before* an Effect E_2 if and only if all of the following apply:

- E_1 is a Memory Effect.
- Any of the following applies:
  - E_1 is DSB-ordered-before E_3.
  - E_1 is Instruction-fetch-barrier-ordered-before E_3.
- There is an IMPLEMENTATION DEFINED instruction sequence beginning with E_3 indirectly leading to the generation of E_2.
- E_2 is a Memory Effect.

If an Effect E_1 is Out-of-band-ordered-before an Effect E_2, then E_1 is Ordered-before E_2.

Note

Arm expects that, in most systems with early acknowledgments, those acknowledgments will come from a point at or after the point that establishes global visibility. This is expected in such systems to enable the acknowledgments to be used as part of the mechanisms to implement the ordering requirements of the Arm memory model.

B2.4.2 Metadata Dependencies

I_{YRYHC} The execution of the following instructions create ordering relations for the purposes of the Arm memory model:

- `IRG`.
- `PACDA`.
- `PACDB`.
- `AUTDA`.
- `AUTDB`

R_{TVHSL} If FEAT_MTE is implemented, the execution of an `IRG Xd,Xn,Xm` instruction creates:

- A Dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.
- If Allocation Tag Access is not prevented for the current Exception level, a Dependency through registers and memory from the Register Read Effect of Xm to Register Write Effect of Xd.

R_{MSWTM} If FEAT_PAuth is implemented, the execution of a `PACDA Xd,Xn` instruction creates:

- A Dependency through registers and memory from the Register Read Effect of Xd to Register Write Effect of Xd.
- If authentication using the APDAKey_EL1 key is enabled, a Dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.

R_{JMNSC} If FEAT_FPAC is implemented, the execution of a `PACDB Xd,Xn` instruction creates:

- A Dependency through registers and memory from the Register Read Effect of Xd to Register Write Effect of Xd.
- If authentication using the APDBKey_EL1 key is enabled, a Dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.

R_{XYRJT} If FEAT_FPAC is implemented, the execution of an `AUTDA Xd,Xn` instruction creates:

- If authentication using the APDAKey_EL1 key is enabled, a Pick dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.
- A Dependency through registers and memory from the Register Read Effect of Xd to Register Write Effect of Xd.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{SPBZX} If FEAT_FPAC is implemented, the execution of an `AUTDB Xd,Xn` instruction creates:

- If authentication using the APDBKey_EL1 key is enabled, a Pick dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.
- A Dependency through registers and memory from the Register Read Effect of Xd to Register Write Effect of Xd.

B2.4.3 Limited ordering regions

FEAT_LOR introduces *limited ordering regions* (LORegions), which allow large systems to perform special load-acquire and store-release instructions that provide order between the memory accesses to a region of the PA map as observed by a set of observers.

LORegions are defined in the Non-secure physical address space. LORegions cannot be defined in the Secure, Realm, or Root physical address spaces.

This feature is supported in AArch64 state only.

B2.4.3.1 Specification of the LORegions

The LORegions are defined in the Non-secure physical memory map using a set of LORegion descriptors. The number of LORegion descriptors is IMPLEMENTATION DEFINED, and can be discovered by reading the LORID_EL1 register.

Each LORegion descriptor consists of:

- A tuple of the following values:
  - A Start Address.
  - An End Address.
  - An LORegion Number.
- Valid bit which indicates whether that LORegion descriptor is valid.

A memory location lies within the LORegion identified by the LORegion Number if the PA lies between the Start Address and the End Address, inclusive. The Start Address must be defined to be aligned to 64KB and the End Address must be defined as the top byte of a 64KB block of memory.

It is permitted for multiple LORegion descriptors with non-overlapping address ranges to be configured with the same LORegion Number.

The LORegion descriptors are programmed using the LORSA_EL1, LOREA_EL1, LORN_EL1, and LORC_EL1 registers in the System register space. These registers describe only memory addresses in the Non-secure memory map. These registers are UNDEFINED if accessed when SCR_EL3.NS == 0.

If a LoadLOAcquire or a StoreLORelease does not match with any LORegion, then:

- The LoadLOAcquire will behave as a Load-Acquire, and will be ordered in the same way with respect to all accesses, independent of their LORegions.
- The StoreLORelease will behave as a Store-Release, and will be ordered in the same way with respect to all accesses, independent of their LORegions.

Note

If no LORegions are implemented, then the LoadLOAcquire and StoreLORelease will therefore behave as a Load-Acquire and Store-Release.

B2.4.4 SVE memory ordering relaxations

I_{CTNGV} The Arm memory model is relaxed for reads and writes generated by SVE load and store instructions.

R_{QLJPC} When two reads generated by SVE vector load instructions have an address dependency, the dependency does not contribute to the dependency-ordered-before relation.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{YMBMZ} When a pair of reads access the same location, and at least one of the reads is generated by an SVE load instruction, for a given observer, the hazard-ordered-before relation does not apply to the pair of reads.

R_{CJHWV} When a single SVE vector store instruction generates multiple writes to the same location, the instruction ensures that these writes appear in the coherence order for that location, in order of increasing vector element number. No other ordering restrictions apply to memory effects generated by the same SVE store instruction.

R_{LVXTJ} If a single SVE load instruction generates multiple reads, the order in which the reads for different elements and registers appear is not architecturally defined.

R_{VMDYZ} If an address dependency exists between two Read Memory and an SVE non-temporal vector load instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

I_{CCWGN} For any SVE load or store instruction that generates multiple single-copy atomic accesses to Normal or Device memory, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic memory element access sizes.

B2.4.5 Streaming SVE mode memory ordering relaxations

R_{HBBTV} If a pair of memory reads access the same location, and at least one of the reads is generated by an Advanced SIMD&FP load instruction, for a given observer, the hazard-ordered-before relation does not apply to the pair of reads when the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level.

See also:

- Streaming SVE mode.

B2.4.6 Ordering rules for GCS

I_{QNKNL} If FEAT_GCS is implemented, see Guarded Control Stack data accesses for relevant ordering rules.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.5 Restrictions on the effects of speculation

A *permitted-speculative-nPC* is one of:

- For any instruction that is neither an Unconditional branch (immediate) nor an exception generating instruction, the PC plus four.
- For any instruction, an exception vector.
- For a direct branch, the PC plus the offset contained in the immediate field of the opcode.
- For an indirect branch, any address.

A *permitted-speculatively-executed-instruction* is the speculative execution of an instruction at a permitted-speculative-nPC where that PC was generated from one of the following:

- An architecturally executed instruction.
- A permitted-speculatively-executed-instruction.

The Arm architecture places certain restrictions on the effects of speculation. These are:

- Each load from a location using a particular VA after an exception return that is a Context Synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- Each load from a location using a particular VA after an exception entry that is a Context Synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception entry.
- Any load from a location using a particular VA before an exception entry that is a Context Synchronization event will not speculatively read data from a store to the same location using the same VA after the exception entry.
- Any load from a location using a particular VA before an exception return that is a Context Synchronization event will not speculatively read data from a store to the same location using the same VA after the exception exit.
- When data is loaded under speculation with a Translation fault, it cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.
- A speculative Explicit Memory Read Effect generated by an instruction (I_2) running in a hardware-defined context ($context_2$) will not speculatively read data from a speculative Explicit Memory Write Effect generated by an instruction (I_1) running in a different hardware-defined context ($context_1$) if I_1 generates an MMU fault Effect instead of the Explicit Memory Write Effect.
- When data is loaded under speculation with a GPC fault, it cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence, and the execution timing of any other instructions in the speculative sequence is not a function of the data loaded under speculation.
- When stage 2 translation is enabled and a stage 1 translation table entry is loaded under speculation with a GPC fault, the output address or Next-level table address from the entry cannot be used to form an address to be used by other fetches in the translation table walk.
- Granule protection checks apply to speculative instruction fetch and speculative execution. Any instruction fetched under speculation with a GPC fault:
  - Cannot cause an update to any architectural or micro-architectural state as a result of speculative execution of the instruction, where the update of the state is dependent on the content of the instruction.
  - Cannot be stored in a cache that is not affected by `DC PAPA` operations.
- If GPCCR_EL3.GPCP == 0, data from a translation table walk for which the granule protection check for the address being accessed has not been architecturally resolved can not be used to form an address for a subsequent read access or for generating syndrome information, until the granule protection check has passed.

  Note

  Permitting read accesses to locations for which the granule protection check has not been architecturally resolved means the GPT does not protect non-idempotent locations from these speculative read operations.

- When data is loaded under speculation from a location that does not have a valid translation for the translation regime being speculated in, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When data is loaded as a result of speculative accesses made after `TLBI` + `DSB` + `ERET` using a translation that was invalidated by the TLBI, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the microarchitectural state.
- Changes to Special-purpose registers can occur speculatively.
- Accesses to Device memory are subject to many restrictions on the effects of Speculation. See Device memory.
- Execute-never controls apply to speculative instruction fetching. See Effects on instruction execution permissions and restrictions on instruction fetch.
- When writing new instructions to memory, there is no requirement for an `SB` instruction to prevent speculative execution of the old code. See Instruction cache maintenance instructions.
- Write speculation that is visible to other observers is prohibited for all memory types.
- A load will not speculatively read data from any store that appears in program order after the load itself.
- From Armv9.4, an instruction (I_1) that is not a permitted-speculatively-executed-instruction must not speculatively perform any function, other than being fetched, decoded or performing any side effects resulting from being fetched or decoded, that alters the microarchitectural state of the system in such a way that some other architecturally executed code (code2) can determine any architectural state, except for any architectural state accessed by I_1 as part of I_1 being fetched and decoded, that could not be determined had I_1 not speculatively executed.

When an architectural or speculative Explicit Memory Write Effect (E_1) to a Location (A_1) is properly synchronized such that a subsequent architectural or speculative instruction fetch (I_1) from A_1 is required to observe E_1, the permitted-speculative-nPC generated from I_1 is permitted to be generated from the opcode before or after the modification of A_1 by E_1.

Note

The prohibition of using data loaded under speculation with faults to form addresses, condition codes or SVE predicate values does not prohibit the use of value predicted data from such locations for such purposes, so long as the training of the data value prediction was from the hardware defined context that is using the prediction. A consequence of this is that training of value prediction cannot be based on data loaded under speculation with a translation or Permission fault.

B2.5.1 AArch64 Speculative Store Bypass Safe

When FEAT_SSBS is implemented, PSTATE.SSBS is a control that can be set by software to indicate whether hardware is permitted to use, in a manner that is potentially speculatively exploitable, a speculative value in a register that has been loaded from memory using a load instruction that speculatively read the location being loaded from, where the entry that is speculatively read is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

When the value of PSTATE.SSBS is 0, hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

Setting PSTATE.SSBS to 0 is deprecated for performance reasons.

When the value of PSTATE.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- If speculation is permitted, then cache timing side channels can lead to addresses being derived using reads of address values that have been speculatively loaded from memory to a register.
- Before the introduction of FEAT_SSBS, the control bit SPSR_ELx.SSBS was RES0, and therefore software written without awareness of FEAT_SSBS is expected to program it to 0. This means that PSTATE.SSBS will not be set, so the PE will not be permitted to use speculative loads with outstanding memory disambiguation issues for any subsequent speculative memory accesses if there is any possibility of those subsequent memory accesses creating a cache timing side channel.

B2.5.2 Definition of exploitative control of speculative execution

The execution of some code (code1) can exploitatively control speculative execution of some other code (code2) if all of the following apply:

- The actions of code1 can influence, in a manner that is not hard-to-determine, the prediction of multi-bit values that determine speculative execution of code2 to cause an irreversible change to the microarchitectural state of the PE that is indicative of some architectural state accessible to the execution context of code2.
- code1 has control in determining the choice of the architectural state that the irreversible change to the microarchitectural state is indicative of.
- The irreversible changes to the microarchitectural state of the PE can be measured by code executing in an execution context other than that of code2 to allow the retrieval of the architectural state in a manner that is not hard-to-determine.

Note

Eviction of entries from one context by another context is not considered exploitative control as long as eviction does not result in other forms of prediction that could be exploitatively controlled.

B2.5.3 Definition of predictive leakage to speculative execution

The execution of some code (code1) can predictively leak to some other code (code2) if all of the following apply:

- The execution of code1 influences, in a manner that is not hard-to-determine, the predictive microarchitectural structures of the implementation that predict multi-bit values, not binary choices, to behave in a way that is indicative of some architectural state accessible to the execution context of code1.
- The predictive microarchitectural structures of the implementation impact the timing of the speculative execution of code2 in a way that enables code2 to recover the architectural state in a manner that is not hard-to-determine.
- Code1 and code2 are not collaborating to communicate using the mechanisms in the previous two bullets.

Note

Mechanisms to prevent the influence and the state recovery being “not hard-to-determine” are left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.

B2.5.4 Restrictions on the effects of speculation from Armv8.5

Note

If SCR_EL3.EEL2 is changed, in order to remove all VMID tagging from Secure EL1 and Secure EL0 entries, each prediction resource should be invalidated by software for:

- Secure EL0 for all ASID and VMID values.
- Secure EL1 for all VMID values.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Further restrictions on speculation are introduced by some additional architectural features as described here.

FEAT_CSV3 introduces these restrictions:

- An attempt to access data under speculation, where the data is inaccessible given the current state of the PE cannot be used to alter the architectural or microarchitectural state in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed through means such as measuring the timing of code. This restriction applies to any of the following cases which make the data inaccessible:
  - reading from a Location with a Permission or Domain fault.
  - directly reading from a system register that is not architecturally readable.
  - reading from a SIMD&FP register, SVE register, or SME register that is not accessible due to traps configured by a more privileged Exception level.
  - indirectly reading from a system register containing a Pointer Authentication Key for which the corresponding Pointer Authentication instructions are disabled.

  Note

  The current state of the PE is defined as the EL, values of all special-purpose registers, and values of all system registers under which the hardware attempts to access the data, which may be on the speculative execution path. If on the speculative execution path, the hardware must respect any changes to the PE state that occur on the speculative path, respecting the same rules that would be required if executing on the non-speculative path. Note that the sequence of instructions on the speculative path may not be the same sequence of instructions that would occur on the non-speculative path as the hardware control flow predictions could cause the two sequences to differ.

  Note

  The following is a list of examples in which use of the inaccessible data could lead to recovery of the value of the inaccessible data:

  - to form an address, generate condition codes, or generate SVE predicate values to be used in the speculative sequence.
  - to form the register value used for the comparison of a compare-and-branch, test-and-branch, or compare-and-swap instruction in the speculative sequence. Note that this is to cover conditional instructions that do not use condition codes to determine their outcomes.
  - by prediction mechanisms such as Cache Prefetch or Data Value predictions.
  - as an input to an instruction in the speculative sequence where the execution timing of the instruction is dependent on the data value.

  Note

  It is permissible for a value of zero to be produced by the load or register read and consumed by an instruction newer than the load in the speculative sequence even if the value of zero results in different execution timing compared to non-zero values.

  Note

  As the effects of speculation are not architecturally visible, this restriction requires that the effect of any speculation cannot give rise to side channels that will leak the values of memory locations, System registers, or Special-purpose registers to a level of privilege that would otherwise not be able to determine those values.

- The value of Allocation tags loaded under speculation as the result of a Tag checked access with a NoTagAccess fault or to Untagged memory cannot influence the execution of the instruction causing the Tag checked access or other instructions in the speculative sequence.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the microarchitectural state.

  Note

  Changes to Special-purpose registers can occur speculatively.

FEAT_CSV2, FEAT_CSV2_1p1, FEAT_CSV2_1p2, FEAT_CSV2_2 and FEAT_CSV2_3 introduce a range of additional restrictions.

If FEAT_CSV2 is implemented:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2), as a result of the behavior of any of the following resources:
  - Branch target prediction based on the branch targets used in context1.
    - This applies to both direct and indirect branches, including return instructions, but excludes the prediction of the direction of a conditional branch.
  - Branch type prediction based on the branch types used in context1, when the hardware predicts a branch of a given type to be a different type of branch, and the type of the branch is used to exploitatively control target prediction of indirect branches within context2. *Branch type* is used to distinguish between:
    - Not a branch, direct branch, indirect branch, and branch and link (or call).
  - Data Value predictions based on data value from execution in context1.

    Note

    `PSTATE.{N,Z,C,V}` values from context1 are not considered a data value for this purpose.

  - Virtual address-based cache prefetch predictions generated as a result of execution in context1, based on, or causing dereference of, data values from memory.
  - Address predictions based on addresses used in context1.
    - This includes any form of memory disambiguation prediction or data forwarding based on partial address match.
  - Any other prediction mechanisms, other than Branch, Data Value, Cache Prefetch predictions, or Address predictions.

  In this definition, the hardware-defined context is determined by:
  - The Exception level.
  - The Security state.
  - When executing at EL1, if EL2 is implemented and enabled in the current Security state, the VMID.
  - When executing at EL0, whether the EL1&0 or the EL2&0 translation regime is in use.
  - When executing at EL0 and using the EL1&0 translation regime, the *address space identifier* (ASID) and, if EL2 is implemented and enabled in the current Security state, the VMID.
  - When executing at EL0 and using the EL2&0 translation regime, the ASID.

  If FEAT_CSV2_2 is implemented, then SCXTNUM_ELx is also part of the hardware-defined context.

  For unprivileged accesses generated by unprivileged memory access instructions, the hardware-defined context associated with the data addresses and data values of the unprivileged accesses is determined as if the access was generated by an instruction executed at EL0.

If either FEAT_CSV2_1p1 or FEAT_CSV2_3 is implemented, code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2) as a result of the behavior of any prediction mechanism based on the branch history used in context1.

If FEAT_CSV2_1p1 is implemented, branch or data values trained from one instruction address cannot exploitatively control, or predictively leak to, the speculative execution of code from a different address.

If FEAT_CSV2_1p2 is implemented, the SCXTNUM_ELx register is implemented, but is not part of the hardware-defined context.

B2.5.5 Restrictions on the effects of speculation from Armv9.5

If FEAT_BTI is implemented:

- Branches predicted or executed under speculation into a Guarded Page that land on instructions other than the intended BTI instructions are also guarded against speculative execution of those instructions.

B2.5.6 Branch history

If FEAT_CLRBHB is not implemented, then the architecture does not define any branch history maintenance instructions for AArch64 state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_CLRBHB is implemented, the `CLRBHB` instruction is available. When the `CLRBHB` instruction is executed, the branch history is cleared for the current context to the extent that branch history information created before the `CLRBHB` instruction cannot be used by code before the `CLRBHB` instruction to exploitatively control the execution of any code in the current context appearing in program order after the instruction.

When FEAT_ECBHB is implemented, the branch history information created in a context before an exception entry to a higher Exception level using AArch64 cannot be used by code before that exception entry to exploitatively control the execution of any code in a different context after the exception entry.

Note

FEAT_CLRBHB and FEAT_ECBHB restrict the use of the branch history to prevent exploitative control. These restrictions apply only to prediction resources that are protected by FEAT_CSV2.

B2.5.7 Execution, data prediction and prefetching restriction System instructions

When FEAT_SPECRES is implemented alone or alongside FEAT_SPECRES2, the System instructions listed in A64 System instructions for prediction restriction prevent predictions based on information gathered from earlier execution within a particular execution context (CTX), from affecting the later speculative execution within that CTX, to the extent that the speculative execution is observable through side-channels.

The prediction restriction System instructions being used by a particular CTX apply to:

- All control flow prediction resources that predict execution addresses.
- Data value prediction.
- Cache allocation prediction.

For these System instructions, the CTX is defined by:

- The Security state.
- The Exception level.
- When executing at EL1, if EL2 is implemented and enabled in the current Security state, the VMID.
- When executing at EL0, whether the EL1&0 or the EL2&0 translation regime is in use.
- When executing at EL0 when using the EL1&0 translation regime, the ASID and, if EL2 is implemented and enabled in the current Security state, the VMID.
- When executing at EL0 when using the EL2&0 translation regime, the ASID.

Note

- The data value prediction applies to all prediction resources that use some form of training to speculate data values as part of an execution.
- The cache allocation applies to all instruction and data caches, and TLB prefetching hardware used by the executing PE that applies to the supplied context.

The context information is passed as a register argument, and is restricted so that:

- Execution of the System instruction at EL0 only applies to the current hardware defined context.
- Execution of the System instruction at EL1only applies to the current VMID and Security state, and does not apply to EL2 or EL3.
- Execution of the System instruction at EL2 can only apply to the current Security state, and does not apply to EL3.

If the System instruction is specified to apply to a combination of Security state and Exception level that is not implemented, or an Exception level which is higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a `NOP`.

When the System instruction is complete and synchronized, no predictions of the restricted type for the affected context are influenced by the execution of the program before the System instruction in a manner that can be observed by the use of any side channels.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Prediction restriction System instructions do not require the invalidation of prediction structures so long as the behavior described for completion is met by an implementation.
- Prediction restriction System instructions are permitted to invalidate more prediction information than is defined by the supplied execution context.

These System instructions are guaranteed to be complete following a DSB that covers both read and write behavior on the same PE that executed the original instruction. A subsequent Context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

In AArch64 state, EL0 access to the System instructions is controlled by:

- When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, SCTLR_EL1.EnRCTX.
- When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, SCTLR_EL2.EnRCTX.

Note

If SCR_EL3.EEL2 is changed, in order to remove all VMID tagging from Secure EL1 and Secure EL0 entries, each prediction resource should be invalidated for:

- Secure EL0 for all ASID and VMID values.
- Secure EL1 for all VMID values.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.6 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring load/store instructions. The memory barriers defined by the Arm architecture provide a range of functionality, including:

- Ordering of load/store instructions.
- Completion of load/store instructions.
- Context synchronization.

The following subsections describe the Arm memory barrier instructions:

- Instruction Synchronization Barrier.
- Data Memory Barrier.
- Speculation Barrier.
- Consumption of Speculative Data Barrier.
- Speculative Store Bypass Barrier.
- Profiling Synchronization Barrier.
- Physical Speculative Store Bypass Barrier.
- Trace Synchronization Barrier.
- Data Synchronization Barrier.
- Shareability and access limitations on the data barrier operations.
- Load-Acquire, Load-AcquirePC, and Store-Release.
- LoadLOAcquire, StoreLORelease.
- Guarded Control Stack Barrier (GCSB).

Note

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are available only when software execution is at EL1 or higher.

`DMB` and `DSB` instructions affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance instructions being executed by the PE.

B2.6.1 Instruction Synchronization Barrier

An `ISB` instruction ensures that all instructions that come after the `ISB` instruction in program order are fetched from the cache or memory after the `ISB` instruction has completed. Using an `ISB` ensures that the effects of context-changing operations executed before the `ISB` are visible to the instructions fetched after the `ISB` instruction. Examples of context-changing operations that require the insertion of an `ISB` instruction to ensure the effects of the operation are visible to instructions fetched after the `ISB` instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the `ISB` instruction take effect only after the `ISB` has been executed.

The pseudocode function for the operation of an `ISB` is `InstructionSynchronizationBarrier()`.

See also Memory barriers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.6.2 Data Memory Barrier

The DMB instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The DMB instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

The full definition of the DMB instruction is covered formally in the Ordering requirements defined by the formal concurrency model and this introduction to the DMB instruction is not intended to contradict that section.

The basic principle of a DMB instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the DMB instruction. The DMB instruction ensures that all affected memory accesses by the PE executing the DMB instruction that appear in program order before the DMB instruction and those which originate from a different PE, to the extent required by the DMB options, which have been observed by the PE before the DMB instruction is executed, are observed by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the DMB instruction are observed by that PE.

The use of a DMB instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The DMB instruction affects only memory accesses and the operation of data cache and unified cache maintenance instructions, see A64 Cache maintenance instructions. It has no effect on the ordering of any other instructions executing on the PE.

The pseudocode function for the operation of a DMB instruction is `DataMemoryBarrier()`.

B2.6.3 Speculation Barrier

An SB instruction is a memory barrier that prevents speculative execution of instructions until after the barrier has completed when those instructions could be observed through side-channels.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation. An example is speculative allocation into any caching structure where the allocation of that entry could indicate any data value present in memory or in the registers.

- Cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be performed when predicting that a instruction that could generate an exception does not generate an exception.

The speculative execution of an SB instruction cannot be as a result of any of the following:

- Control flow speculation.
- Data value speculation.
- Can be as a result of predicting that an instruction that could generate an exception does not generate an exception.

An SB instruction can complete when all of the following apply:

- It is known that it is not speculative, or it is speculative only as a result of either:
  - Speculating that an instruction that could generate an exception does not generate an exception.
  - Speculating past the point in the Execution stream where a precise asynchronous exception is taken.
- All the data values generated by instructions appearing in program order before the SB instruction are architecturally resolved, and so are not speculative.

Note

The SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched, so long as the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after the SB instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.6.4 Consumption of Speculative Data Barrier

The `CSDB` instruction is a memory barrier instruction that controls speculative execution arising from data value prediction.

Any instruction, other than a branch instruction, that appears in program order after the CSDB cannot be speculatively executed using the results of any of the following predictions if those predictions come from instructions that appear in program order before the CSDB and have not been architecturally resolved:

- Data value predictions of any instructions.
- `PSTATE.{N,Z,C,V}` predictions of any instructions other than conditional branch instructions appearing in program order before the `CSDB` that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

Note

For purposes of the definition of CSDB, `PSTATE.{N,Z,C,V}` and SVE predication state are not considered data values. This definition permits:

- Control flow speculation before and after the `CSDB` instruction.
- Speculative execution of conditional data processing instructions after the `CSDB` instruction, unless they use the results of data value, SVE predication state, or `PSTATE.{N,Z,C,V}` predictions of instructions appearing in program order before the `CSDB` instruction that have not been architecturally resolved.

B2.6.5 Speculative Store Bypass Barrier

The `SSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `SSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the `SSBB` instruction.

B2.6.6 Profiling Synchronization Barrier

The `PSB` 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.

If the Statistical Profiling Extension is not implemented, this instruction executes as a `NOP`.

B2.6.7 Physical Speculative Store Bypass Barrier

The `PSSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `PSSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the `PSSBB` instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

B2.6.8 Trace Synchronization Barrier

The `TSB` instruction is a barrier instruction that preserves the relative order of accesses to System registers due to trace operations and other accesses to the same registers.

A trace operation is an operation of the trace unit generating trace for an instruction when FEAT_TRF is implemented and enabled.

A `TSB` instruction is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more Context synchronization events are required to ensure that `TSB` instruction is executed in the necessary order.

If trace is generated between a Context synchronization event and a `TSB` operation, these trace operations may be reordered with respect to the `TSB` operation, and therefore may not be synchronized.

The following situations are synchronized using a `TSB` operation:

- A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  - A is executed in program order before a Context synchronization event C.
  - C appears in program order before a `TSB` operation T.
  - B is executed in program order after T.
- A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  - A is executed in program order before a Context synchronization event C1.
  - C1 appears in program order before `TSB` operation T.
  - T is executed in program order before a second Context synchronization event C2.
  - B is executed in program order after C2.

A `TSB` operation is not needed to ensure a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:

- A is executed in program order after a Context synchronization event C.
- B is executed in program order before C.

If FEAT_TRBE is implemented, the requirements in this section are extended.

See Synchronization and the Trace Buffer Unit.

The pseudocode function for the operation of a `TSB` instruction is `TraceSynchronizationBarrier()`.

B2.6.9 Data Synchronization Barrier

A `DSB` instruction is a memory barrier that ensures that memory accesses that occur before the `DSB` instruction have completed before the completion of the `DSB` instruction. In doing this, it acts as a stronger barrier than a `DMB` and all ordering that is created by a `DMB` with specific options is also generated by a `DSB` with the same options.

Execution of a `DSB` instruction:

- At EL2 ensures that any memory accesses caused by Speculative translation table walks from the EL1&0 translation regime have been observed.
- At EL3 ensures that any memory accesses caused by speculative translation table walks from the EL2, EL1&0 or EL2&0 translation regimes have been observed.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see Out-of-context translation regimes.

A DSB instruction executed by a PE completes when all of the following are complete for the set of observers in the required shareability domain:

- If the required access types of the DSB is reads, all of the following:
  - All Explicit Memory Read Effects generated by instructions in program order before the DSB.
  - If FEAT_GCS is implemented, all GCS Memory Read Effects that appear in program order before a GCSB instruction that appears in program order before the DSB.
  - If FEAT_MTE2 is implemented, all Implicit Tag Memory Read Effects generated by instructions in program order before the DSB.
- If the required access types of the DSB is writes, all of the following:
  - All Explicit Memory Write Effects generated by instructions in program order before the DSB.
  - If FEAT_GCS is implemented, all GCS Memory Write Effects that appear in program order before a GCSB instruction that appears in program order before the DSB.
- If the required access types of the DSB is reads and writes, all of the following:
  - All Explicit Memory Effects generated by instructions in program order before the DSB.
  - All Implicit TTD Memory Effects generated by instructions in program order before the DSB.
  - All Implicit Instruction Memory Read Effects generated by instructions in program order before the DSB.
  - All cache maintenance instructions in program order before the DSB.
  - If FEAT_MTE2 is implemented, all Implicit Tag Memory Read Effects generated by instructions in program order before the DSB.
  - If FEAT_SPMU is implemented, all Direct System Register Write Effects to a System PMU register generated by instructions in program order before the DSB.
  - If FEAT_SPE is implemented, all PSB CSYNC instructions in program order before the DSB.
  - If FEAT_XS is implemented and the DSB has the nXS qualifier, all of the following:
    - All TLBInXS maintenance operations generated by AArch64 TLB maintenance instructions with the nXS qualifier in program order before the DSB.
    - If HCRX_EL2.FnXS is 1, all TLBInXS maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL1 in program order before the DSB.
  - If FEAT_XS is not implemented or the DSB does not have the nXS qualifier, all of the following:
    - All TLBI maintenance operations generated by TLB maintenance instructions in program order before the DSB.
  - If FEAT_HDBSS is implemented and the DSB is executed at EL2 or EL3, all HDBSS Memory Write Effects generated by instructions in program order before the DSB.
  - If FEAT_TRBE is implemented, all TSB CSYNC instructions in program order before the DSB.
  - All CFP RCTX, COSP RCTX, CPP RCTX, DVP RCTX instructions in program order before the DSB.
  - If FEAT_MTE_ASYNC is implemented, all Indirect System Register Write Effects to the Tag Fault Status Register accessible at ELx generated by instructions in program order before the DSB.
  - If FEAT_GCS is implemented, all GCS Memory Effects that appear in program order before a GCSB instruction that appears in program order before the DSB.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes other than:

- Being fetched from memory and decoded, or any change to architectural or microarchitectural state resulting from being fetched or decoded.
- Indirectly or directly reading any of the following, so long as the reading of these resources does not cause a change in architectural state:
  - General-purpose registers
  - SIMD&FP registers
  - The Stack Pointer register
  - The Program Counter
  - If FEAT_SVE or FEAT_SME is implemented, SVE scalable vector registers or SVE predicate registers
  - If FEAT_SVE is implemented, the First Fault Register
  - If FEAT_SME is implemented, the ZA storage

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If FEAT_SME2 is implemented, the ZT0 register
  - Special-purpose registers
  - System registers other than CNTPCTSS_EL0 and CNTVCTSS_EL0
- If FEAT_ETS2 is not implemented, translating any virtual addresses of instructions that generate Explicit Memory Effects.

If FEAT_MTE_ASYNC is implemented, on completion of a `DSB` instruction with the LD qualifier, or neither the LD nor ST qualifier, all updates to TFSR_ELx.TFy or TFSRE0_EL1.TFy due to Tag Check Faults caused by accesses generated by instructions occuring in program order before the `DSB` will be complete. For more information on FEAT_MTE_ASYNC, see The Memory Tagging Extension.

When FEAT_XS is implemented and HCRX_EL2.FnXS is 1, an AArch64 `DSB` instruction executed at EL1 or EL0 behaves in the same way as the corresponding `DSB` instruction with the nXS qualifier executed at EL1 or EL0.

A `DSB` instruction ordered after a direct write to a System PMU register does not complete until all observers observe the direct write. A Context synchronization event is required to create the order between the direct write and the `DSB` instruction.

If FEAT_TRBE is implemented, the requirements in this section are extended. See Trace synchronization and memory barriers.

The pseudocode function for the operation of a `DSB` is `DataSynchronizationBarrier()`.

See also:

- Memory barriers.
- Ordering and completion of TLB maintenance instructions.

B2.6.10 Shareability and access limitations on the data barrier operations

The `DMB` and `DSB` instructions take an argument that specifies:

- The shareability domain over which the instruction must operate. This is one of:
  - Full system.
  - Outer Shareable.
  - Inner Shareable.
  - Non-shareable.

  Full system applies to all the observers in the system and, as such, encompasses the Inner and Outer Shareable domains of the processor.

  Note

  The distinction between Full system and Outer Shareable is applicable only for Normal Non-cacheable memory accesses and Device memory accesses.

- The accesses for which the instruction operates. This is one of:
  - Read and write accesses, both before and after the barrier instruction.
  - Write accesses only, before and after the barrier instruction.
  - Read accesses before the barrier instruction, and read and write accesses after the barrier instruction.

    Note

    This form of a `DMB` or `DSB` instruction can be described as a load-load/store barrier.

For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier or Data Synchronization Barrier.

Table B2-1 shows how these options are encoded in the `<option>` field of the instruction:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table B2-1 Encoding of the DMB and DSB <option> parameter

Accesses		Shareability domain			
Before the barrier	**After the barrier**	**Full system**	**Outer Shareable**	**Inner Shareable**	**Non-shareable**
Reads and writes	Reads and writes	`SY`	`OSH`	`ISH`	`NSH`
Writes	Writes	`ST`	`OSHST`	`ISHST`	`NSHST`
Reads	Reads and writes	`LD`	`OSHLD`	`ISHLD`	`NSHLD`

Note

Table B2-1 specifies the memory ordering requirements of DMB and DSB based on the `<option>` parameter. DSB has additional requirements with respect to execution of instructions appearing in program order after the DSB as described in the section Data Synchronization Barrier. These additional requirements create the effect of additional memory ordering beyond what is specified in the table.

See the instruction descriptions for more information:

- DMB.
- DSB.

Note

`ISB` also supports an optional limitation argument that can contain only one value that corresponds to full system operation, see ISB.

B2.6.11 Load-Acquire, Load-AcquirePC, and Store-Release

Arm provides a set of instructions with Acquire semantics for loads, and Release semantics for stores. These instructions support the *Release Consistency sequentially consistent* (RCsc) model. In addition, FEAT_LRCPC provides Load-AcquirePC instructions. The combination of Load-AcquirePC and Store-Release can be use to support the weaker *Release Consistency processor consistent* (RCpc) model.

The full definitions of the Load-Acquire and Load-AcquirePC instructions are covered formally in the Ordering requirements defined by the formal concurrency model. This introduction to the Load-Acquire and Load-AcquirePC instructions is not intended to contradict that section.

The basic principle of both Load-Acquire and Load-AcquirePC instructions is to introduce order between:

- The memory access generated by the Load-Acquire or Load-AcquirePC instruction.
- The memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction, such that the memory access generated by the Load-Acquire or Load-AcquirePC instruction is observed by each PE to the extent that the PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction are observed by that PE to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire or Load-AcquirePC instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the Ordering requirements defined by the formal concurrency model and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the following:

- A set of memory accesses, RWx, that are generated by the PE executing the Store-Release instruction and that appear in program order before the Store-Release instruction, together with those that originate from a different PE to the extent that the PE is required to observe them coherently, observed by the PE before executing the Store-release.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The memory access generated by the Store-Release (Wrel), such that all of the memory accesses, RWx, are observed by each PE to the extent that the PE is required to observe those accesses coherently, before Wrel is observed by that PE to the extent that the PE is required to observe that access coherently.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

Where a Load-Acquire appears in program order after a Store-Release, the memory access generated by the Store-Release instruction is observed by each PE to the extent that PE is required to observe the access coherently, before the memory access generated by the Load-Acquire instruction is observed by that PE, to the extent that the PE is required to observe the access coherently. In addition, the use of a Load-Acquire, Load-AcquirePC or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral arrival order.

Load-Acquire, Load-AcquirePC and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element.

Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements.

A Store-Release Exclusive instruction has the release semantics only if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit `DMB` instruction.

B2.6.12 LoadLOAcquire, StoreLORelease

For each PE, the Non-secure physical memory map is divided into a set of LORegions using a table that is held within the PE. Any PA in the Non-secure memory map can be a member of one LORegion. If a PA is assigned to more than one LORegion, then an implementation might treat it as if it has been assigned to fewer LORegions than that have been specified. If a PA is not in the Non-secure physical memory map, then that PA cannot be a member of any LORegion. For more information, see Limited ordering regions.

FEAT_LOR provides a set of instructions with Acquire semantics for loads, and Release semantics for stores that apply in relation to the defined LORegions. The new variants of the Load-Acquire and Store-Release instructions are LoadLOAcquire and StoreLORelease. See LoadLOAcquire/StoreLORelease.

For all memory types, these instructions have the following ordering requirements:

- LoadLOAcquire has the same semantics as Load-Acquire except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See Load-Acquire, Load-AcquirePC, and Store-Release.
- StoreLORelease has the same semantics as Store-Release except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See Load-Acquire, Load-AcquirePC, and Store-Release.

In addition, for accesses to Memory-mapped peripherals:

- LoadLOAcquire has the same semantics as Load-Acquire except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See Load-Acquire, Load-AcquirePC, and Store-Release.
- StoreLORelease has the same semantics as Store-Release except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See Load-Acquire, Load-AcquirePC, and Store-Release.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit `DMB` instruction.

B2.6.13 Guarded Control Stack Barrier (GCSB)

The `GCSB` instruction is a barrier instruction that generates a GCSB event. The GCSB event applies ordering requirements between general load/store accesses and Guarded Control Stack data accesses. For more information, see Guarded Control Stack data access behaviors.

If FEAT_GCS is not implemented, this instruction executes as a `NOP`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.7 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. The Arm architecture defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- Introduction to caches.
- Memory hierarchy.
- Application level access to functionality related to caches
- Implication of caches for the application programmer.
- Prefetching into cache.

B2.7.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access. Caching takes account of two principles of locality:

Spatial locality

An access to one Location is likely to be followed by accesses to adjacent Locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. The Arm architecture permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

B2.7.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an Armv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure B2-1 shows an example of such a system in an Armv8-A system that supports virtual addressing.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B2-1 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the processing element, as shown in Figure B2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches that are located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU), Point of Coherency (PoC), Point of Persistence (PoP), and Point of Deep Persistence (PoDP).

For more information, including the definitions of PoU, PoC, PoP, and PoDP, see About cache maintenance in AArch64 state.

If FEAT_MTE2 is implemented, the behavior of cache maintenance instructions is modified. For more information, see Allocation Tags.

B2.7.2.1 The cacheability and shareability memory attributes

Cacheability and shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

Cacheability This attribute defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

Shareability This attribute defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about Cacheability and Shareability, see Memory types and attributes.

B2.7.3 Application level access to functionality related to caches

As indicated in About the Application level programmers' model, the application level corresponds to execution at EL0. The architecture defines a set of cache maintenance instructions that software can use to manage cache coherency. Software executing at a higher Exception level can enable use of some of this functionality from EL0, as follows:

When the value of SCTLR_EL1.UCI is 1

Software executing at EL0 can access:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The data cache maintenance instructions, `DC CVAU`, `DC CVAC`, `DC CVAP`, `DC CVADP`, and `DC CIVAC`. See The data cache maintenance instruction (DC).
- The instruction cache maintenance instruction `IC IVAU`. See The instruction cache maintenance instruction (IC).

Attempted execution of these instructions might generate a Permission fault as described in Permission fault.

When the value of SCTLR_EL1.UCT is 1

Software executing at EL0 can access the cache type register. See CTR_EL0.

When the value of SCTLR_EL1.DZE is 1

Software executing at EL0 can access the data cache zero instruction `DC ZVA`. See Data cache zero instruction.

The SCTLR_EL1.{UCI, UCT, DZE} control fields are accessible only by software executing at EL1 or higher.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the controls {UCI, UCT and DZE} are found in SCTLR_EL2.

This functionality is UNDEFINED at EL0 when the value of the corresponding SCTLR_EL1 control field is 0.

B2.7.4 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

B2.7.4.1 Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using cache maintenance instructions to manage the coherency issues in software. See Application level access to functionality related to caches.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see Non-shareable Normal memory and Shareable, Inner Shareable, and Outer Shareable Normal memory.

Note

The performance of these hardware coherency mechanisms is highly IMPLEMENTATION SPECIFIC. In some implementations, the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Not all these mechanisms are directly available to software operating at EL0 and might involve interaction with software operating at a higher Exception level.

B2.7.4.2 Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last Context Synchronization event on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory. In the absence of a Context Synchronization event, there is no limit on the number of times such an instruction might be executed without being refetched from memory.

The Arm architecture requires the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory. A write has been made coherent with an instruction fetch of a shareability domain when:

CTR_EL0.{DIC, IDC} == {0, 0} The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {0, 1} The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 1} The write is complete for the shareability domain.

Note

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
- For an implementation:
  - When CTR_EL0.DIC == 0, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the IC IVAU instructions of a different core.
  - When CTR_EL0.DIC == 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

The full definition of the ordering implications of CTR_EL0.{DIC, IDC} is covered formally in the Ordering requirements defined by the formal concurrency model and the information in this section is not intended to contradict that section.

If software requires coherency between instruction execution and memory, it must manage this coherency using Context Synchronization events and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
        DSB ISH             ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
        DSB ISH             ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- If this sequence is not executed between writing data to a location and executing the instruction at that location, the lack of coherency between instruction caches and memory means that the instructions that are executed might be the old instruction or the updated instruction, and which is used can arbitrarily vary during execution. It must not be assumed by software, before the synchronization sequence is executed, that when the updated instruction has been seen, the old instruction will not be seen again.
- For Non-cacheable or Write-Through accesses, the clean data cache by VA instruction is not required. However, the invalidate instruction cache instruction is required because the Armv8 AArch64 architecture allows Non-cacheable accesses to be held in an instruction cache. See Non-cacheable accesses and instruction caches.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See Concurrent modification and execution of instructions.
- The system software controls whether these cache maintenance instructions are available to the application level by setting SCTLR_EL1.UCI.

B2.7.5 Prefetching into cache

The Arm architecture provides memory system hints `PRFM`, `PRFUM`, `LDNP`, and `STNP` that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring the data or instruction locations into caches.

The Prefetch instructions are hints, and so implementations can treat them as `NOP`s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is taken using an SError exception. For more information, see ISS encoding for an exception from a Data Abort.

`PrefetchHint {}` defines the prefetch hint types.

The `Hint_Prefetch()` function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as prefetching the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

For more information on `PRFM`, `PRFUM` and load/store instructions that provide hints to the memory system, see Prefetch memory and Load/store SIMD and floating-point non-temporal pair. For more information on `SVE PRF*` instructions that provide hints to the memory system, see Predicated non-contiguous element accesses.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.8 Alignment support

This section describes alignment support. It contains the following subsections:

- Instruction alignment.
- Alignment of data accesses.

B2.8.1 Instruction alignment

A64 instructions must be word-aligned.

Attempting to fetch an instruction from a misaligned location results in a PC alignment fault. See PC alignment checking.

B2.8.2 Alignment of data accesses

When instructions that load or store single or multiple registers are executed, if the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated if any of the following apply:

- For Load-Exclusive, Store-Exclusive, and Atomic instructions, the address that is accessed is not aligned to the overall access size.
- For all other instructions, the address that is accessed is not aligned to the size of the data element being accessed.

If FEAT_LSE2 is implemented, Load-Acquire/Store-Release instructions without exclusive or atomic behavior generate an Alignment fault if all of the following apply:

- Not all bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes.
- The value of SCTLR_ELx.nAA applicable to the current Exception level is 0.

For an unaligned access to any type of Device memory:

- If the memory location cannot support unaligned accesses then an Alignment fault is generated.
- If the memory location supports unaligned accesses, then it is IMPLEMENTATION DEFINED whether the access generates an Alignment fault if the location would not generate an Alignment fault if the same access were made to Normal memory.

B2.8.2.1 Unaligned accesses to Normal memory

The behavior of unaligned accesses to Normal memory is dependent on all of the following:

- The instruction causing the memory access.
- The memory attributes of the accessed memory.
- The value of SCTLR_ELx.{A, nAA}.
- Whether or not FEAT_LSE2 is implemented.

B2.8.2.1.1 Load or Store of Single or Multiple registers

For all instructions that load or store single or multiple registers, but not Load-Exclusive, Store-Exclusive, Load-Acquire/Store-Release, Atomic, and `SETG*` Memory Copy and Memory Set instructions, if the address that is accessed is not aligned to the size of the data element being accessed, then:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

- An unaligned access is performed.
- If FEAT_LSE2 is not implemented, the access is not guaranteed to be single-copy atomic except at the byte access level.
- If FEAT_LSE2 is implemented:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, the memory access is single-copy atomic. For `LDNP`, `LDP`, or `STP` instructions, the entire memory access will be single-copy atomic.

— If all the bytes of the memory accessed do not lie within a 16-byte quantity aligned to 16 bytes, or the access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, the access is not guaranteed to be single-copy atomic except at the byte access level.

For these instructions, the definition of an unaligned access is based on the size of the accessed elements, not the overall size of the memory access. This affects SIMD and SVE element and structure loads and stores, and also load/store pair instructions.

For predicated SVE vector element and structure load or store instructions, alignment checks are based on the memory element access size, not on the vector element size.

For predicated SVE vector element and structure load or store instructions, *Inactive elements* cannot cause an Alignment fault.

For unpredicated SVE vector register load or store instructions, the base address is checked for 16-byte alignment.

For unpredicated SVE predicate register load or store instructions, the base address is checked for 2-byte alignment.

B2.8.2.1.2 Load-Exclusive/ Store-Exclusive and Atomic instructions

If FEAT_LSE2 is not implemented and the value of SCTLR_ELx.A applicable to the current Exception level is 0, Load-Exclusive/Store-Exclusive and Atomic instructions, including those with acquire or release semantics, generate an alignment fault when the address being accessed is not aligned to the overall size of the access.

If FEAT_LSE2 is implemented, then:

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, an unaligned access is performed.
- If all the bytes of the memory access do not lie within a 16-byte quantity aligned to 16-bytes, or the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  — An unaligned access is performed meeting all of the semantics of the instruction.
  — An Alignment fault is generated.

Where memory access is performed, then it is single-copy atomic.

For these instructions, the definition of an unaligned access is based on the overall access size.

If FEAT_LS64 is implemented, when a single-copy atomic 64-byte instruction accesses a memory location that is not aligned to 64 bytes, an Alignment fault always occurs, regardless of the value of SCTLR_ELx.A.

If FEAT_THE is implemented, Read-Check-Write instructions, `RCW`, and Read-Check-Write Software instructions, `RCWS`, generate an Alignment fault if the address being accessed is not aligned to the overall access size regardless of the value of SCTLR_ELx.A.

B2.8.2.1.3 Non-atomic Load-Acquire/Store-Release instructions

For Load-Acquire/Store-Release instructions that do not have exclusive or atomic behaviors, when the address being accessed is not aligned to the size of the data element being accessed and the value of SCTLR_ELx.A applicable to the current Exception level is 0:

If FEAT_LSE2 is not implemented, then these instructions generate an Alignment fault.

If FEAT_LSE2 is implemented, then:

- If the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  — An unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.
  — An Alignment fault is generated.
- If all of the bytes of the memory access do not lie within a 16-byte quantity aligned to 16 bytes and SCTLR_ELx.nAA applicable to the current Exception level is 1, then an unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In this case, when an access is performed it meets all the semantics of the instruction, but the architecture does not define the order of the different transactions of the access defined by the single instructions relative to each other.

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, an unaligned access meeting all the semantics of the instruction is performed.

Note

- Unaligned accesses typically take additional cycles to complete compared to a naturally-aligned access.
- An operation that is not single-copy atomic above the byte level can abort on any memory access that it makes and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the page boundary.

B2.8.2.1.4 Memory Copy and Memory Set instructions

For `SETG*` instructions:

- There is an alignment check regardless of the value of SCTLR_ELx.A.
- If Xn is not a multiple of 16, an Alignment fault is generated.
- If Xd is not aligned to a multiple of 16, an Alignment fault is generated.

For more information, see the individual `SETG*` instruction descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.9 Endian support

General description of endianness in the Arm architecture describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- Instruction endianness.
- Data endianness.
- Endianness of memory-mapped peripherals.

B2.9.1 General description of endianness in the Arm architecture

This section describes only memory addressing and the effects of endianness for data elements up to quadwords of 128 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure B2-2 shows, for big-endian and little-endian memory systems, the relationship between:

- The quadword at address A.
- The doubleword at address A and A+8.
- The words at addresses A, A+4, A+8, and A+12.
- The halfwords at addresses A, A+2, A+4, A+6, A+8, A+10, A+12, and A+14.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, A+7, A+8, A+9, A+10, A+11, A+12, A+13, A+14, and A+15.

The terms in Figure B2-2 have the following definitions:

B_A Byte at address A.

HW_A Halfword at address A.

MSByte Most significant byte.

LSByte Least significant byte.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure B2-2 Endianness relationships

The big-endian and little-endian mapping schemes determine the order in which the bytes of a quadword, doubleword, word, or halfword are interpreted. For example, a load of a word from address `0x1000` always results in an access to the bytes at memory locations `0x1000`, `0x1001`, `0x1002`, and `0x1003`. The endianness mapping scheme determines the significance of these 4 bytes.

B2.9.2 Instruction endianness

A64 instructions have a fixed length of 32 bits and are always little-endian.

B2.9.3 Data endianness

SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0. When the Effective value of HCR_EL2.{E2H,TGE} is {1, 1}, the control is from SCTLR_EL2.E0E.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for SIMD element and data structure loads and stores. For more information, see Endianness in SIMD operations.

Note

This means the Armv8 architecture introduces a requirement for 128-bit endian conversions.

B2.9.3.1 Instructions to reverse bytes in a general-purpose register, a SIMD and floating-point register, or an SVE scalable vector register

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table B2-2 shows the instructions that provide this functionality:

Table B2-2 Byte reversal instructions

Function	Instructions	Notes
Reverse bytes in 32-bit word or words[a]	REV32	For use with general-purpose registers
Reverse bytes in whole register	REV	For use with general-purpose registers
Reverse bytes in 16-bit halfwords	REV16	For use with general-purpose registers
Reverse elements in doublewords, vector	REV64	For use with SIMD and floating-point registers
Reverse elements in words, vector	REV32	For use with SIMD and floating-point registers
Reverse elements in halfwords, vector	REV16	For use with SIMD and floating-point registers
Reverse bytes/halfwords/words within elements, predicated	REVB, REVH, REVW	For use with SVE scalable vector registers

a. Can operate on multiple words.

B2.9.3.2 Endianness in SIMD operations

SIMD element load/store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used to load and store data correctly in both big-endian and little-endian systems.

For example:

```
LD1 {V0.4H}, [X1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure B2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

Figure B2-3 SIMD byte order example

The `BigEndian()` pseudocode function determines the current endianness of the data.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()`‘ functions are defined in A-profile Architecture Pseudocode.

B2.9.3.3 Endianness in SVE operations

R_{VDGQK} Rules on byte and element order of SIMD load and store instructions apply to SVE load and store instructions.

I_{RFQJP} Additional rules apply to the data endianness of memory accesses performed by SVE load and store instructions.

R_{CNKCL} For predicated SVE vector element and structure load and store instructions, an endianness conversion is performed using the memory element access size. The size of the vector element is not used in endianness conversion.

R_{QHXPL} For unpredicated SVE vector register load and store instructions, the vector byte elements are transferred in increasing element number order without any endianness conversion.

R_{RWLXY} For unpredicated SVE predicate register load and store instructions, each 8 bits from the predicate are transferred as a byte in increasing element number order without any endianness conversion.

R_{YGSBQ} When an SVE load instruction is executed, endianness conversion occurs before any sign-extension or zero-extension into a vector element.

R_{KYRQW} When an SVE store instruction is executed, endianness conversion occurs after any truncation from the vector element to the memory element access size.

B2.9.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a Cross Trigger Interface, see About the External Debug Registers.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see System Level Implementation of the Generic Timer.
- A memory-mapped register interface to the Performance Monitors, see Recommended External Interface to the Performance Monitors.
- A memory-mapped register interface to the Activity Monitors, see Recommended External Interface to the Activity Monitors.
- A memory-mapped register interface to the Memory System Resource Partitioning and Monitoring architecture (MPAM), see the *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification (ARM IHI 0099).*
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4.*
- The memory-mapped register interface to an Arm trace component. See, for example, the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4.*

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.10 Memory types and attributes

The ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- Normal memory.
- Device memory.
- Memory access restrictions.

B2.10.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform Speculative data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- An Explicit Memory Write Effect to a memory location with the Normal attribute completes in finite time.
- Explicit Memory Write Effects to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes. For the purposes of this requirement, the endpoint for a location in Conventional memory is the PoC.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See Multi-register loads and stores that access Normal memory.
- Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction.

Note

Write speculation that is visible to other observers is prohibited for all memory types.

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.

The following sections describe the other attributes for Normal memory:

- Shareable Normal memory.
- Non-shareable Normal memory.
- Cacheability attributes for Normal memory.

See also:

- Multi-register loads and stores that access Normal memory.
- Atomicity in the Arm architecture.
- Memory barriers.
- Concurrent modification and execution of instructions.

B2.10.1.1 Shareable Normal memory

A Normal memory location has a Shareability attribute that is one of:

- Inner Shareable, meaning it applies across the Inner Shareable shareability domain.
- Outer Shareable, meaning it applies across both the Inner Shareable and the Outer Shareable shareability domains.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, which hardware must enforce. They do not affect the coherency requirements of instruction fetches, see Synchronization and coherency issues between data and instruction accesses.

Note

- System designers can use the shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different processing elements.
- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

B2.10.1.1.1 Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is a member of only a single Inner Shareability domain.
- Each observer is a member of only a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Note

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The details of the use of the shareability attributes are system-specific. Example B2-1 shows how they might be used.

Example B2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable shareability domain.

For shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

B2.10.1.2 Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

B2.10.1.3 Cacheability attributes for Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information, see Cacheability, cache allocation hints, and cache transient hints.

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint, this does not affect the behavior of accesses to that location.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For other cases, see Mismatched memory attributes.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.
- For accesses to Normal memory that is Non-cacheable, a DMB instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations.

Note

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.
- An implementation might not have any outer cache.

Example B2-2, Example B2-3, and Example B2-4 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

Note

- L1 cache is the level closest to the PE, see Memory hierarchy.
- When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example B2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 and L2 cache.
- The Outer cacheability attribute applied to L3 cache.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example B2-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

Example B2-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

B2.10.1.4 Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more SVE or Advanced SIMD&FP registers from an Exception level, there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

B2.10.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Armv8 Device memory are:

Gathering Identified as G or nG, see Gathering.

Reordering Identified as R or nR, see Reordering.

Early Write Acknowledgement

Identified as E or nE, see Early Write Acknowledgement.

The Armv8 Device memory types are:

Device-nGnRnE Device non-Gathering, non-Reordering, No Early Write Acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that Speculative accesses to Device-GRE memory is forbidden.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

  The following exceptions to this apply:

  — Reads generated by the SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — For reads, including hardware speculation, that are performed by an SVE unpredicated load instruction, all of the following are true:

    — For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction, any byte in the window can be accessed by the instruction.

    — All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction.

  — For reads, including hardware speculation, that are performed by an SVE predicated load instruction that is not a non-temporal load, all of the following are true:

    — For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by an *Active element* of the instruction, any byte in the window can be accessed by the instruction.

    — All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes that contains at least 1 byte that is explicitly accessed by an *Active element* of the instruction.

  — For reads, including hardware speculation, that are performed by an SVE predicated non-temporal load instruction from memory locations with the Gathering attributes, all of the following are true:

    — For any 128-byte window aligned to 128 bytes containing at least 1 byte that is explicitly accessed by an *Active element* of the instruction, any byte in the window can be accessed by the instruction.

    — All bytes accessed by the instruction are in a 128-byte window aligned to 128 bytes that contains at least 1 byte that is explicitly accessed by an *Active element* of the instruction.

  — The architecture permits a Memory Copy and Memory Set `CPY*` instruction to perform speculative reads of any memory location, even those marked as Device, within a 64-byte quantity, aligned to 64 bytes, of a location that is within the range [Xs] to [Xs+Xn-1].

  — For Device memory with the Gathering attribute, reads generated by the `LDNP` instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction. See Properties of single-copy atomic accesses.

  — An LDRAA or LDRAB instruction that fails the pointer authentication check and loads from a location in Device memory is permitted to cause one read access to that location if all of the other requirements for accessing that Device location are met.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

— An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception, the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program defines only a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.

— Write speculation that is visible to other observers is prohibited for all memory types.

- An Explicit Memory Write Effect to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit Memory Write effect by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Explicit Memory Write Effects to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two Explicit Memory Write Effects of Device memory type to the same location might be merged before they reach the endpoint, unless both Explicit Memory Write Effects have the non-Gathering attribute or there is an ordered-before relationship between the two Explicit Memory Write Effects.
- For accesses to any Device memory type, a `DMB` instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations.
- If a memory location is not capable of supporting unaligned memory accesses, then an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- If a memory location is capable of supporting unaligned memory accesses, and such a memory location is marked as Device, then it is IMPLEMENTATION DEFINED whether an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as execute-never for all Exception levels.

Note

This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as execute-never for all Exception levels is a programming error.

Note

In the EL1&0 translation regime in systems where HCR_EL2.TGE == 1 and HCR_EL2.DC == 0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes ESR_EL2.ISS[24] to be 0.

See also Memory access restrictions.

The memory types for translation table walks cannot be defined as any Device memory type within the TCR_ELx. For the EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR_EL2.PTW bit is 1, a stage 2 Permission fault is generated if a first stage translation table walk is made to any Device memory type.

Note

In general, making a translation table walk to any Device memory type is the result of a programming error.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

B2.10.2.1 Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

Note

This also applies to writebacks from the cache, whether caused by a Natural eviction or as a result of a cache maintenance instruction.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All accesses occur at their single-copy atomic sizes, except that there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by multi-register load/store instructions that generate more than one single-copy atomic access. See Multi-register loads and stores that access Device memory.

Gathering between two accesses, where one access is barrier-ordered-before the other access, is not permitted.

Gathering between memory accesses generated by Memory Copy and Memory Set instructions is always permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

Note

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a `DMB` or `DSB` barrier that affects both of the accesses.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  - The accesses are not generated by a Store-Release instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.10.2.2 Reordering

In the Device memory attribute:

R Indicates that the location has the Reordering attribute. Accesses to the location can be reordered within the same rules that apply to accesses to Normal Non-cacheable memory. All memory types with the Reordering attribute have the same ordering rules as accesses to Normal Non-cacheable memory, see Ordering relations.

nR Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

Note

Some interconnect fabrics, such as PCIe, perform very limited reordering, which is not important for the software usage. It is outside the scope of the Arm architecture to prohibit the use of a non-Reordering memory type with these interconnects.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

Note

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the DMB instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.
- The non-Reordering property is only required by the architecture to apply the order of arrival of accesses to a single memory-mapped peripheral of an IMPLEMENTATION DEFINED size, and is not required to have an impact on the order of observation of memory accesses to SDRAM. For this reason, there is no effect of the non-Reordering attribute on the ordering relations between accesses to different locations described in Ordering relations as part of the formal definition of the memory model. It does have an effect on the Peripheral arrival order described in Peripheral arrival order.
- If the same memory location is mapped with different aliases, and different attribute values, these are a type of mismatched attribute. The different attributes could be:
  - A different Reordering attribute value.
  - A different Device memory attribute value.
  - When FEAT_XS is implemented, a different XS attribute value.

For information about the effects of accessing memory with mismatched attributes, see Mismatched memory attributes.

An implementation:

- Is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.
- Is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The non-Reordering attribute has no effect on the ordering of cache maintenance instructions, even if the memory location specified in the instruction has the non-Reordering attribute.

The non-Reordering attribute has no effect on the memory accesses caused by Memory Copy and Memory Set instructions.

B2.10.2.3 Early Write Acknowledgement

In the Device memory attribute:

E Indicates that the location has the Early Write Acknowledgement attribute.

nE Indicates that the location has the No Early Write Acknowledgement attribute.

If the No Early Write Acknowledgement attribute is assigned for a Device memory location:

- For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgment of a write comes from the endpoint, it is guaranteed that:
  - Only the endpoint of the write access returns a write acknowledgment of the access.
  - No earlier point in the memory system returns a write acknowledgment.
- For memory system endpoints where the system architecture in which the PE is operating does not require that acknowledgment of a write comes from the endpoint, the acknowledgment of a write is not required to come from the endpoint.

Note

A write with the No Early Write Acknowledgement attribute assigned for a Device memory location is not expected to generate an abort in any situation where the equivalent write to the same location without the No Early Write Acknowledgement attribute assigned does not generate an abort.

This means that a `DSB` barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system if that is required by the system architecture.

Peripherals are an example of system endpoints that require that the acknowledgment of a write comes from the endpoint.

Note

- The Early Write Acknowledgement attribute only affects where the endpoint acknowledgment is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.
- The areas of the physical memory map for which write acknowledgment from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgment from the endpoint.
- Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgment from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgment from the endpoint.
- For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a `DSB` instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgment from the endpoint.

B2.10.2.4 Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register, the order in which the registers are accessed is not defined by the architecture. This applies even to accesses to any type of Device memory.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For all instructions that load or store one or more Advanced SIMD&FP or SVE registers, and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions, even for access to any type of Device memory.

The architecture permits that the non-speculative execution of an instruction that loads or stores more than one general-purpose or SIMD and floating-point register might result in repeated accesses to the same address, even if the resulting accesses are to any type of Device memory.

B2.10.2.5 SVE loads and stores that access Device memory

R_{XLLJZ} All rules applying to Device memory accesses by Advanced SIMD and floating-point load and store instructions apply to Device memory access by SVE load and store instructions.

I_{YHWJT} Additional rules apply to Device memory access by SVE load and store instructions.

R_{NYMWH} When an SVE vector prefetch instruction is executed, any resulting Read Memory is guaranteed not to access Device memory.

R_{SBHLD} When an SVE Non-fault vector load is executed, or when any element from a First-fault load except the *First active element* attempts to access Device memory, the resulting Read Memory will not access Device memory.

R_{TMVNR} When an SVE Non-fault vector load instruction is executed, an attempt by any *Active element* to access Device memory is suppressed and reported in the FFR.

R_{SFBKQ} When an SVE First-fault vector load instruction is executed, any Read Memory performed for the *First active element* can access Device memory.

R_{BHNQN} When an SVE First-fault vector load instruction is executed, an attempt by any *Active element* other than the *First active element* to access Device memory is suppressed and is reported in the FFR.

R_{QBLMZ} Any access to Device memory performed by an SVE load or store instruction is relaxed such that it might behave as if:

- The memory has the Gathering attribute, regardless of the configured value of the nG attribute.
- The memory has the Reordering attribute, regardless of the configured value of the nR attribute.
- The memory has the Early Acknowledgment attribute, regardless of the configured value of the nE attribute.

Whether attributes are classified as mismatched is determined strictly by the memory attributes derived from the translation table entry.

B2.10.2.6 Streaming SVE mode loads and stores that access Device memory

R_{XHFBX} When the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, any access to Device memory performed by an Advanced SIMD&FP load/store instruction is relaxed such that it might behave as if:

- The Gathering attribute is 1, regardless of the configured value of the nG attribute.
- The Reordering attribute is 1, regardless of the configured value of the nR attribute.
- The Early Acknowledgement attribute is 1, regardless of the configured value of the nE attribute.

Whether attributes are classified as mismatched is determined strictly by the memory attributes derived from the translation table entry.

See also:

- Streaming SVE mode.

B2.10.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For two explicit Memory Read effects to any two adjacent bytes in memory, p and $p+1$, generated by the same instruction, and for two explicit Memory Write effects to any two adjacent bytes in memory, p and $p+1$, that are generated by the same instruction:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The bytes *p* and *p+1* must have the same memory type and Shareability attributes, otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LD1`, `ST1`, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.

— Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes *p* and *p+1*.

For the permitted CONSTRAINED UNPREDICTABLE behavior, see Crossing a page boundary with different memory types or Shareability attributes.

- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross an address boundary that corresponds to the smallest implemented translation granule, then behavior is CONSTRAINED UNPREDICTABLE, and Crossing a peripheral boundary with a Device access describes the permitted behaviors. For this reason, it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

Note

— The boundary referred to is between two Device memory regions that are both of the size of the smallest implemented translation granule and aligned to the size of the smallest implemented translation granule.

— This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

— Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.11 Mismatched memory attributes

Memory attributes are controlled by privileged software. For more information, see The AArch64 Virtual Memory System Architecture.

Physical memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type: Device-nGnRnE, Device-nGnRE, Device-nGRE, Device-GRE or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.
- When FEAT_XS is implemented, XS attribute.

Collectively these are referred to as memory attributes.

If FEAT_MTE2 is implemented, accesses to a location which use a common definition of the memory attributes but the Tagged attribute of that location differs do not cause a mismatched access to occur.

If FEAT_MTE_PERM is implemented, the NoTagAccess attribute does not factor into the determination of mis-matched attributes.

Note

In this document, the terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes, the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of Speculative read accesses.
- Prohibition on Gathering.
- Prohibition on reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
   - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   - Either:
     - All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.

2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
   - Any Device memory type.
   - Inner Non-cacheable, Outer Non-cacheable Normal memory.

   In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   - Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
   - Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting DMB barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
   - Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
   - After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
   - Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a DMB barrier instruction, with scope that applies to the common shareability of the accesses, between any accesses to the same cacheable Location that use different attributes.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see Ordering and completion of data and instruction cache instructions.

   Note

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different shareability attributes, then uniprocessor semantics, ordering, and coherency are guaranteed only if:
   - Software running on a PE cleans and invalidates a Location from cache before and after each read or write to that Location by that PE.
   - A DMB barrier with scope that covers the full shareability of the accesses is placed between any accesses to the same memory Location that use different attributes.

   Note

   The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

Note

As described in Non-cacheable accesses and instruction caches, a non-cacheable access is permitted to be cached in an instruction cache, despite the fact that a non-cacheable access is not permitted to be cached in a unified cache. Despite this, when cacheable and non-cacheable aliases exist for memory which is executable, these must be treated as mismatched aliases to avoid coherency issues from the data or unified caches that might hold entries that will be brought into the instruction caches.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.12 Synchronization and semaphores

Armv8 provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal memory and to any type of Device memory.

Note

Use of the Armv8 synchronization primitives scales for multiprocessing system designs.

Table B2-3 shows the synchronization primitives and the associated `CLREX` instruction.

Table B2-3 Synchronization primitives and associated instruction, A64 instruction set

Transaction size	Additional semantics	Load-Exclusive[a]	Store-Exclusive[a]	Other[a]
Byte	-	`LDXRB`	`STXRB`	-
	Load-Acquire/Store-Release	`LDAXRB`	`STLXRB`	-
Halfword	-	`LDXRH`	`STXRH`	-
	Load-Acquire/Store-Release	`LDAXRH`	`STLXRH`	-
Register[b]	-	`LDXR`	`STXR`	-
	Load-Acquire/Store-Release	`LDAXR`	`STLXR`	-
Pair[b]	-	`LDXP`	`STXP`	-
	Load-Acquire/Store-Release	`LDAXP`	`STLXP`	-
None	Clear-Exclusive	-	-	`CLREX`

a. Instruction in the A64 instruction set.

b. A register instruction operates on a doubleword if accessing an X register, or on a word if accessing a W register A pair instruction operates on two doublewords if access X registers, or on two words if accessing W registers.

Except for the row showing the `CLREX` instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the Write Memory succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see Marking and the size of the marked memory block. A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- Exclusive access instructions and Non-shareable memory locations.
- Exclusive access instructions and Shareable memory locations.
- Marking and the size of the marked memory block.
- Context switch support.
- Load-Exclusive and Store-Exclusive instruction usage restrictions.
- Use of WFE and SEV instructions by spin-locks.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.12.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the shareability attribute is Non-shareable, the exclusive access instructions rely on a *local Exclusives monitor*, or *local monitor*, that marks the address of the Explicit Memory Read effect generated by any Load-Exclusive instruction executed by the PE.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Explicit Memory Write effect of the Store-Exclusive instruction is the same as the address that has been marked in the monitor due to the execution of an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place, the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state, the monitor is *set.*

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Open Access state, the monitor is *clear*.

The Store-Exclusive instruction defines the register to which the status value is returned.

When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure B2-4 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address `x` used for the operation.

Figure B2-4 Local monitor state machine diagram

For more information about marking, see Marking and the size of the marked memory block.

Note

For the local monitor state machine, as shown in Figure B2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

B2.12.1.1 Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure B2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure B2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure B2-4 must not indefinitely delay forward progress of execution.

B2.12.2 Exclusive access instructions and Shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, which marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in Exclusive access instructions and Non-shareable

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

memory locations, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:

— Updating memory.

— Returning a status value of 0.

The local monitor can ignore accesses from other PEs in the system.

- A *global monitor* that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:

  — Can hold at least one marked block.

  — Maintains a state machine for each marked block it can hold.

  Note

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, see Load-Exclusive and Store-Exclusive instruction usage restrictions.

Note

The global monitor can either reside within the PE, or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

If FEAT_MTE2 is implemented, it is IMPLEMENTATION DEFINED whether explicit accesses to Allocation Tags are monitored by a global monitor. For more information, see The Memory Tagging Extension.

Note

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all Arm PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

The architecture only requires that Conventional memory mapped in this way supports this functionality.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive or a Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

  If the IMPLEMENTATION DEFINED MMU fault is generated for the EL1&0 translation regime then:

  — If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.

  — Otherwise, the fault is a second stage fault and the exception is taken to EL2.

  The priority of this fault is IMPLEMENTATION DEFINED.
- The instruction is treated as a `NOP`.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN. In this case, if the store exclusive instruction is a store exclusive pair of 64-bit quantities, then the two quantities being stored might not be stored atomically.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by Arm PEs is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

B2.12.2.1 Operation of the global Exclusives monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

Note

The global monitor supports only a single outstanding exclusive access to shareable memory per PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:

  — A status value of 0 is returned to a register to acknowledge the successful store.

  — The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.

  — If the address accessed is marked for exclusive access in the global monitor state machine for any other PE, then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:

  — A status value of 1 is returned to a register to indicate that the store failed.

  — The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:

  — If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.

  — If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means that it responds to:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Accesses generated by PE(n).
- Accesses generated by the other observers in the shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

A global monitor:

- In the Exclusive Access state is *set*.
- In the Open Access state is *clear*.

B2.12.2.1.1 Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see Mechanisms for entering a low-power state.

Figure B2-5 shows the state machine for PE(n) in a global monitor.

‡`StoreExcl`(Marked_address,!n) clears the monitor only if the `StoreExcl` updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking, see Marking and the size of the marked memory block.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For the global monitor state machine, as shown in Figure B2-5:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure B2-5 shows only how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

B2.12.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size 2^a bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4-512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where *a* is 4.
- 512 words in an implementation where *a* is 11.

For example, in an implementation where *a* is 4, a successful `LDXRB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR_EL0. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

B2.12.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

B2.12.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDXP`/`STXP` pair or a `LDXR`/`STXR` pair, executed as part of a loop that contains only a single Load-Exclusive and Store-Exclusive pair.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To support different implementations of these functions, software must follow the notes and restrictions given here.

The following notes describe the use of a `LoadExcl`/`StoreExcl` instruction pair, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table B2-3. In this context, a `LoadExcl`/`StoreExcl` pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target VA of a `StoreExcl` is different from the VA of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, the status value returned by the `StoreExcl` is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses.

  - The data at the address accessed by the `LoadExcl`, and at the address accessed by the `StoreExcl`, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` are executed with the same VA.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a `StoreExcl` instruction is the same as the transaction size of the preceding `LoadExcl` instruction executed in that thread. If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes.

  - The block of data of the size of the larger of the transaction sizes used by the `LoadExcl`/`StoreExcl` pair at the address accessed by the `LoadExcl`/`StoreExcl` pair, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` have the same transaction size.
- An implementation of the `LoadExcl` and `StoreExcl` instructions can require that, in any thread of execution, the `StoreExcl` instruction accesses the same number of registers as the preceding `LoadExcl` instruction executed in that thread. If the `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. As a result, software can rely on an `LoadExcl`/`StoreExcl` pair to eventually succeed only if they access the same number of registers. For more information, see CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the Tag Checked property of a memory access due to a `StoreExcl` instruction is the same as the Tag Checked property of a memory access by the preceding `LoadExcl` instruction, unless FEAT_MTE_STORE_ONLY is implemented and the preceding `LoadExcl` is not Tag Checked only due to Store-only Tag Checking being enabled. If the Tag Checked property of memory accesses due to a `LoadExcl`/`StoreExcl` pair in the same thread of execution is not compatible, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of such a `LoadExcl`/`StoreExcl` pair, and fail for other instances of such a `LoadExcl`/`StoreExcl` pair.

  - The data at the address accessed by the `LoadExcl`/`StoreExcl` pair is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the memory is accessed with a compatible Tag Checked property.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_LSE is not implemented, `LoadExcl`/`StoreExcl` loops are guaranteed to make forward progress only if, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, the software meets all of the following conditions. If FEAT_LSE is implemented, software can maximize the likelihood that `LoadExcl`/`StoreExcl` loops make forward progress, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, by meeting all of the following conditions:

  1 Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory effects, software prefetches, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, `ISB` barriers, indirect branches, or Branch with Link instructions.

  2 Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:

  — There are no stores or `PRFM` or `RPRFM` instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive. This includes VA aliases to those addresses.

  — There are no loads or prefetches to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.

  — There are no other Store-Exclusive instructions to any other address.

  — There are no direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, indirect branches, or Branch with Link instructions.

  — All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

  The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the `LoadExcl`/`StoreExcl` loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:

  — Performing stores to a PA covered by the Exclusives monitor.

  — Prefetching with intent to write to a PA covered by the Exclusives monitor.

  — Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.

  — Executing instruction cache invalidate all instructions.

  — Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.

  — Executing TLB maintenance to a PA covered by the Exclusives monitor.

- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.

- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.

- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.

- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:

  — The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

  Note

  This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes.

  — The data at the address accessed by the `StoreExcl` is UNKNOWN.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Another bullet point in this list covers the case where the memory attributes of a `LoadExcl`/`StoreExcl` pair differ as a result of using different VAs with different attributes that point to the same PA.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same shareability domain as the PE executing the cache maintenance instruction, as determined by the shareability domain of the address being maintained.

  Note

  Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in one Security state cannot cause a denial of service on a PE in another Security state.

- If the mapping of the VA to PA is changed between the `LoadExcl` instruction and the `StoreExcl` instruction, and the change is performed using a break-before-make sequence as described in Using break-before-make when updating translation table entries, if the `StoreExcl` is performed after another write to the same PA as the `StoreExcl`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `StoreExcl` will not pass its monitor check.

  Note

  Arm expects that, in many implementations, either:

  — The TLB invalidation will clear either the local or global monitor.

  — The PA will be checked between the `LoadExcl` and `StoreExcl`.

- The Exclusive Access state for an address accessed by a PE can be lost as a result of a `PRFM PST*` or `RPRFM` instruction to the same PA executed by another PE. This means that a very high rate of repeated `PRFM PST*` or `RPRFM` accesses to a memory location might impede the forward progress of another PE.
- If FEAT_MTE2 is implemented, and if a Tag Unchecked store exclusive instruction would not perform the store and return a status value of 1, it is CONSTRAINED UNPREDICTABLE whether:

  — The instruction is a Tag Checked access,

  — The instruction is an Tag Unchecked access.

  For more information, see The Memory Tagging Extension.

Note

In the event of repeatedly-contending `LoadExcl`/`StoreExcl` instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

B2.12.5.1 CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers

As stated in this section, an implementation can require that the instructions of a Load-Exclusive/Store-Exclusive pair access the same number of registers. In such an implementation, this means behavior is CONSTRAINED UNPREDICTABLE if, in a single thread of execution, either:

- An `LDXP` instruction of two 32-bit quantities is followed by an `STXR` instruction of one 64-bit quantity at the same address.
- An `LDXR` instruction of one 64-bit quantity is followed by an `STXP` instruction of two 32-bit quantities at the same address.

In these cases, the CONSTRAINED UNPREDICTABLE behavior must be one of:

- The `STXP` or `STXR` instruction generates an external Data Abort.
- The `STXP` or `STXR` instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `0b110101`.
- The `STXP` or `STXR` instruction always fails, returning a status of 1.
- The `STXP` or `STXR` instruction always passes, returning a status of 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- This STXP or STXR instruction has the same pass or fail behavior that it would have had if the instruction had used the same size and number of registers as the preceding LDXR or LDXP instruction.

B2.12.6 Use of WFE and SEV instructions by spin-locks

Armv8 provides Wait For Event, Send Event, and Send Event Local instructions, WFE, SEV, and SEVL, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in Wait for Event. However, in Armv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note

This is equivalent to issuing an SEVL instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an SEV instruction after clearing a spinlock.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part C
The AArch64 Instruction Set

Chapter C1
The A64 Instruction Set

This chapter describes the A64 instruction set. It contains the following sections:

- About the A64 instruction set.
- Structure of the A64 assembler language.
- Address generation.
- Instruction aliases.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.1 About the A64 instruction set

The A64 instruction set is the instruction set supported in the AArch64 Execution state. All A64 instructions have a width of 32 bits.

The following chapters contain alphabetical lists of A64 instructions:

- A64 Base Instruction Descriptions lists all of:
  - Branch instructions, exception generating instructions, system instructions.
  - Load, store, and data-processing instructions associated with the general-purpose registers.
- A64 Advanced SIMD and Floating-point Instruction Descriptions lists the load, store, and data-processing instructions associated with Advanced SIMD and floating-point support.
- SVE Instruction Descriptions lists the load, store, and data-processing instructions associated with SVE support.
- SME Instruction Descriptions lists the load, store, and data-processing instructions associated with SME support.

When an instruction supports more than one syntax, each syntax is an *instruction variant*. Instruction variants can occur because of differences in:

- The size or format of the operands.
- The register file used for the operands.
- The addressing mode used for load/load/store memory operands.

Instruction variants might also arise as the result of other factors.

Instruction variants are described in the instruction description for the individual instructions.

A64 instruction set encoding describes the A64 encoding structure. A64 instructions have a regular bit encoding structure:

- 5-bit register operand fields at fixed positions within the instruction. For general-purpose register operands, the values 0-30 select one of 31 registers. The value 31 is used as a special case that can:
  - Indicate use of the current stack pointer, when identifying a load/store base register or in a limited set of data-processing instructions. See The stack pointer registers.
  - Indicate the value zero when used as a source register operand.
  - Indicate discarding the result when used as a destination register operand.
- For Advanced SIMD and floating-point register access and SVE scalable vector register access, a 5-bit register operand field selects one of 32 registers of the appropriate type.
- Certain Advanced SIMD, SVE, and SME instructions have a register operand field smaller than 5 bits when specifying an indexed vector element or multi-vector operand, which restricts the number of vector registers that can be accessed by those operands.
- For SME instructions that access ZA tiles, ZA tile slices, or ZA array vectors, see ZA storage.
- Immediate bits that provide constant data-processing values or address offsets are placed in contiguous bitfields. Some computed values in instruction variants use one or more immediate bitfields together with the secondary encoding bitfields.

All encodings that are not fully defined are described as unallocated. An attempt to execute an unallocated instruction is UNDEFINED, unless the behavior is otherwise defined in this Manual.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2 Structure of the A64 assembler language

The following sections describe the A64 assembler syntax:

- General requirements.
- Common syntax terms.
- Instruction Mnemonics.
- Condition code.
- Register names.

C1.2.1 General requirements

The letter `W` denotes a general-purpose register holding a 32-bit word, and `X` denotes a general-purpose register holding a 64-bit doubleword.

An A64 assembler recognizes both uppercase and lowercase variants of the instruction mnemonics and register names, but not mixed case variants. An A64 disassembler can output either uppercase or lowercase mnemonics and register names. Program and data labels are case-sensitive.

The A64 assembly language does not require the # character to introduce constant immediate operands, but an assembler must allow immediate values introduced with or without the # character.

In Example C1-1, the sequence // is used as a comment leader and A64 assemblers are encouraged to accept this syntax.

C1.2.2 Common syntax terms

The following syntax terms are used frequently throughout the A64 instruction set description.

`UPPER` Text in upper-case letters is fixed. Text in lower-case letters is variable. This means that register name `Xn` indicates that the `X` is required, followed by a variable register number, for example `X29`.

`< >` Any text enclosed by angle braces, < >, is a value that the user supplies. Subsequent text might supply additional information.

`{ }` Any item enclosed by curly brackets, { }, is optional. A description of the item and how its presence or absence affects the instruction is normally supplied by subsequent text. In some cases curly braces are actual symbols in the syntax, for example when they surround a register list. These cases are called out in the surrounding text.

`[ ]` Any items enclosed by square brackets, [ ], constitute a list of alternative characters. A single one of the characters can be used in that position and the subsequent text describes the meaning of the alternatives. In some case the square brackets are part of the syntax itself, such as addressing modes or vector elements. These cases are called out in the surrounding text.

`a|b` Alternative words are separated by a vertical bar, |, and can be surrounded by parentheses to delimit them. For example, `U(ADD|SUB)W` represents `UADDW` or `USUBW`.

`±` This indicates an optional + or - sign. If neither is used then + is assumed.

`uimmn` An *n*-bit unsigned, positive, immediate value.

`simmn` An *n*-bit two's complement, signed immediate value, where *n* includes the sign bit.

`SP` See Register names.

`Wn` See Register names.

`WSP` See Register names.

`WZR` See Register names.

`Xn` See Register names.

`XZR` See Register names

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2.3 Instruction Mnemonics

The A64 assembly language overloads instruction mnemonics and distinguishes between the different forms of an instruction based on the operand types. For example, the following `ADD` instructions all have different opcodes. However, the programmer must remember only one mnemonic, as the assembler automatically chooses the correct opcode based on the operands. The disassembler follows the same procedure in reverse.

Example C1-1 ADD instructions with different opcodes

```
ADD W0, W1, W2          // add 32-bit register
ADD X0, X1, X2          // add 64-bit register
ADD X0, X1, W2, SXTW    // add 64-bit extended register
ADD X0, X1, #42         // add 64-bit immediate
```

C1.2.4 Condition code

The A64 ISA has some instructions that set Condition flags or test Condition codes or both. For information about instructions that set the Condition flags or use the condition mnemonics, see Condition flags and related instructions.

Table C1-1 shows the available Condition codes.

Table C1-1 Condition codes

cond	Mnemonic	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal or unordered	Z == 0
0010	CS or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Ordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C ==1 && Z == 0
1001	LS	Unsigned lower or same	Less than or equal	!(C ==1 && Z ==0)
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 && N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z == 0 && N == V)
1110	AL	Always	Always	Any
1111	NV[b]	Always	Always	Any

a. Unordered means at least one NaN operand.

b. The Condition code `NV` exists only to provide a valid disassembly of the `0b1111` encoding, otherwise its behavior is identical to `AL`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2.5 SVE Condition code aliases

The SVE and SME assembler syntax defines an alternative set of condition code aliases for use with AArch64 conditional instructions, as follows:

Table C1-2 shows the available SVE Condition code aliases.

Table C1-2 SVE Condition codes

cond	Mnemonic	SVE alias	Meaning	Condition flags
0000	EQ	NONE	All *Active elements* were FALSE or there were no *Active elements*.	Z == 1
0001	NE	ANY	An *Active element* was TRUE.	Z == 0
0010	CS or HS	NLAST	The *Last active element* was FALSE or there were no *Active elements*.	C == 1
0011	CC or LO	LAST	The *Last active element* was TRUE.	C == 0
0100	MI	FIRST	The *First active element* was TRUE.	N == 1
0101	PL	NFRST	The *First active element* was FALSE or there were no *Active elements*.	N == 0
0110	VS	-	CTERM comparison failed, but end of partition reached.	V == 1
0111	VC	-	CTERM comparison succeeded, or end of partition not reached.	V == 0
1000	HI		An *Active element* was TRUE, but the *Last active element* was FALSE.	C ==1 && Z == 0
1001	LS	PLAST	The *Last active element* was TRUE, or all *Active elements* were FALSE, or there were no *Active elements*.	C ==0 || Z ==1
1010	GE	TCONT	CTERM termination condition not detected.	N == V
1011	LT	TSTOP	CTERM termination condition detected.	N != V

C1.2.6 Register names

See:

- General-purpose register file and zero register and stack pointer.
- Advanced SIMD and floating-point register file.
- Advanced SIMD and floating-point scalar register names.
- SIMD vector register names.
- SIMD vector element names.
- For SVE register names, see Z0-Z31, P0-P15, and FFR, First Fault Register.
- For SME ZA storage, see ZA array vector access and ZA tile access.
- For SME2 ZT0 storage, see ZT0.

C1.2.6.1 General-purpose register file and zero register and stack pointer

The 31 general-purpose registers in the general-purpose register file are named R0-R30 and encoded in the instruction register fields with values 0-30. In a general-purpose register field the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position.

When the registers are used in a specific instruction variant, they must be qualified to indicate the operand data size, 32 bits or 64 bits, and the data size of the instruction.

When the data size is 32 bits, the lower 32 bits of the register are used and the upper 32 bits are ignored on a read and cleared to zero on a write.

Table C1-3 shows the qualified names for registers, where *n* is a register number 0-30.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C1-3 Naming of general-purpose registers, the zero register, and the stack pointer

Name	Size	Encoding	Description
Wn	32 bits	0-30	General-purpose register 0-30
Xn	64 bits	0-30	General-purpose register 0-30
WZR	32 bits	31	Zero register
XZR	64 bits	31	Zero register
WSP	32 bits	31	Current stack pointer
SP	64 bits	31	Current stack pointer

This list gives more information about the instruction arguments shown in Table C1-3:

- The names Xn and Wn both refer to the same general-purpose register, Rn.
- There is no register named W31 or X31.
- The name SP represents the stack pointer for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as a read or write of the current stack pointer. When instructions do not interpret this operand encoding as the stack pointer, use of the name SP is an error.
- The name WSP represents the current stack pointer in a 32-bit context.
- The name XZR represents the zero register for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as returning zero when read or discarding the result when written. When instructions do not interpret this operand encoding as the zero register, use of the name XZR is an error.
- The name WZR represents the zero register in a 32-bit context.
- The architecture does not define a specific name for general-purpose register R30 to reflect its role as the link register on procedure calls. However, an A64 assembler must always use W30 and X30 for this purpose, and additional software names might be defined as part of the Procedure Call Standard, see *Procedure Call Standard for the Arm 64-bit Architecture*.

C1.2.6.2 Advanced SIMD and floating-point register file

The 32 registers in the Advanced SIMD and floating-point register file, V0-V31, hold floating-point operands for the scalar floating-point instructions, and both scalar and vector operands for the Advanced SIMD instructions. When they are used in a specific instruction form, the names must be further qualified to indicate the data shape, that is the data element size and the number of elements or lanes within the register. A similar requirement is placed on the general-purpose registers. See General-purpose register file and zero register and stack pointer.

Note

The data type is described by the instruction mnemonics that operate on the data. The data type is not described by the register name. The data type is the interpretation of bits within each register or vector element, whether these are integers, floating-point values, polynomials, or cryptographic hashes.

C1.2.6.3 Advanced SIMD and floating-point scalar register names

Advanced SIMD and floating-point instructions that operate on scalar data only access the lower bits of an Advanced SIMD and floating-point register. The unused high bits are ignored on a read and cleared to 0 on a write.

Table C1-4 shows the qualified names for accessing scalar Advanced SIMD and floating-point registers. The letter *n* denotes a register number between 0 and 31.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C1-4 Advanced SIMD and floating-point scalar register names

Size	Name
8 bits	`Bn`
16 bits	`Hn`
32 bits	`Sn`
64 bits	`Dn`
128 bits	`Qn`

C1.2.6.4 SIMD vector register names

If a register holds multiple data elements on which arithmetic is performed in a parallel, SIMD manner, then a qualifier describes the vector shape. The vector shape is the element size and the number of elements or lanes. If the element size in bits multiplied by the number of lanes does not equal 128, then the upper 64 bits of the register are ignored on a read and cleared to zero on a write.

Table C1-5 shows the SIMD vector register names. The letter *n* denotes a register number between 0 and 31.

Table C1-5 SIMD vector register names

Shape	Name
8 bits × 8 lanes	`Vn.8B`
8 bits × 16 lanes	`Vn.16B`
16 bits × 4 lanes	`Vn.4H`
16 bits × 8 lanes	`Vn.8H`
32 bits × 2 lanes	`Vn.2S`
32 bits × 4 lanes	`Vn.4S`
64 bits × 1 lane	`Vn.1D`
64 bits × 2 lanes	`Vn.2D`

C1.2.6.5 SIMD vector element names

Appending a constant, zero-based element index to the register name inside square brackets indicates that a single element from a SIMD and floating-point register is used as a scalar operand. The number of lanes is not represented, as it is not encoded in the instruction and can only be inferred from the index value.

Table C1-6 shows the vector register names and the element index. The letter *i* denotes the element index.

Table C1-6 Vector register names with element index

Size	Name
8 bits	`Vn.B[i]`
16 bits	`Vn.H[i]`
32 bits	`Vn.S[i]`
64 bits	`Vn.D[i]`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An assembler must accept a fully qualified SIMD register name if the number of lanes is greater than the index value. See SIMD vector register names. For example, an assembler must accept all of the following forms as the name for the 32-bit element in bits [63:32] of the SIMD and floating-point register V9:

```
V9.S[1]     //standard disassembly
V9.2S[1]    //optional number of lanes
V9.4S[1]    //optional number of lanes
```

Note

The SIMD and floating-point register element name `Vn.S[0]` is not equivalent to the scalar SIMD and floating-point register name `Sn`. Although they represent the same bits in the register, they select different instruction encoding forms, either the vector element or the scalar form.

C1.2.6.5.1 SIMD vector register list

Where an instruction operates on multiple SIMD&FP or SVE registers, for example vector load/store structure and table lookup operations, the registers are specified as a list enclosed by curly braces. This list consists of either a sequence of registers separated by commas, or a register range separated by a hyphen. The registers must be numbered in increasing order, modulo 32, in increments of one. The hyphenated form is preferred for disassembly if there are more than two registers in the list and the register numbers are increasing by 1. The following examples are equivalent representations of a set of four registers `V4` to `V7`, each holding four lanes of 32-bit elements:

```
{ V4.4S - V7.4S }              //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S } //alternative representation
```

C1.2.6.5.2 SIMD vector element list

Registers in a list can also have a vector element form. For example, the `LD4` instruction can load one element into each of four registers, and in this case the index is appended to the list as follows:

```
{ V4.S - V7.S }[3]                //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S }[3] //alternative with optional number of lanes
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.3 Address generation

The A64 instruction set supports 64-bit virtual addresses (VAs). The valid VA range is determined by the following factors:

- The size of the implemented virtual address space.
- *Memory Management Unit* (MMU) configuration settings.
- *Memory Protection Unit* (MPU) configuration settings.

Limits on the VA size mean that the most significant bits of the virtual address do not hold valid address bits. These unused bits can hold:

- A tag, see Address tagging.
- If FEAT_PAuth is implemented, a Pointer authentication code (PAC), see Pointer authentication.

For more information on memory management and address translation, see The AArch64 Virtual Memory System Architecture.

C1.3.1 Register indexed addressing

The A64 instruction set allows a 64-bit index register to be added to the 64-bit base register, with optional scaling of the index by the access size. Additionally it allows for sign-extension or zero-extension of a 32-bit value within an index register, followed by optional scaling.

C1.3.2 PC-relative addressing

The A64 instruction set has support for position-independent code and data addressing:

- PC-relative literal loads have an offset range of ± 1MB.
- Process state flag and compare based conditional branches have a range of ± 1MB. Test bit conditional branches have a restricted range of ± 32KB.
- Unconditional branches, including branch and link, have a range of ± 128MB.

PC-relative load/store operations, and address generation with a range of ± 4GB can be performed using two instructions.

C1.3.3 Load/store addressing modes

Load/store addressing modes in the A64 instruction set require a 64-bit base address from a general-purpose register X0-X30 or the current stack pointer, `SP`, with an optional immediate or register offset. Table C1-7 shows the assembler syntax for the complete set of load/store addressing modes.

Table C1-7 A64 Load/store addressing modes

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	`[base{, #0}]`	-	-
Base plus offset	`[base{, #imm}]`	`[base, Xm{, LSL #imm}]`	`[base, Wm, (S|U)XT(X|W){#imm}]`
Pre-indexed	`[base, #imm]!`	-	-
Post-indexed	`[base], #imm`	`[base], Xm`[a]	-
Literal (PC-relative)	`label`	-	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

a. The post-indexed by register offset mode can be used with the SIMD load/store structure instructions described in Load/store Advanced SIMD. Otherwise the post-indexed by register offset mode is not available.

Some types of load/store instruction support only a subset of the load/store addressing modes listed in Table C1-7. Details of the supported modes are as follows:

- Base plus offset addressing means that the address is the value in the 64-bit base register plus an offset.
- Pre-indexed addressing means that the address is the sum of the value in the 64-bit base register and an offset, and the address is then written back to the base register.
- Post-indexed addressing means that the address is the value in the 64-bit base register, and the sum of the address and the offset is then written back to the base register.
- Literal addressing means that the address is the value of the 64-bit program counter for this instruction plus a 19-bit signed word offset. This means that it is a 4 byte aligned address within ±1MB of the address of this instruction with no offset. Literal addressing can be used only for loads of at least 32 bits and for prefetch instructions. The PC cannot be referenced using any other addressing modes. The syntax for labels is specific to individual toolchains.
- An immediate offset can be unsigned or signed, and scaled or unscaled, depending on the type of load/store instruction. When the immediate offset is scaled it is encoded as a multiple of the transfer size, although the assembly language always uses a byte offset, and the assembler or disassembler performs the necessary conversion. The usable byte offsets therefore depend on the type of load/store instruction and the transfer size. Table C1-8 shows the offset and the type of load/store instruction.

Table C1-8 Immediate offsets and the type of load/store instruction

Offset bits	Sign	Scaling	Write-Back	Load/store type
0	-	-	-	Exclusive/acquire/release
7	Signed	Scaled	Optional	Register pair
9	Signed	Unscaled	Optional	Single register
12	Unsigned	Scaled	No	Single register

- A register offset means that the offset is the 64 bits from a general-purpose register, Xm, optionally scaled by the transfer size, in bytes, if `LSL #imm` is present and where `imm` must be equal to $\log_2$(transfer_size). The `SXTX` extend/shift option is functionally equivalent to `LSL`, but the `LSL` option is preferred in source code.
- An extended register offset means that offset is the bottom 32 bits from a general-purpose register Wm, sign-extended or zero-extended to 64 bits, and then scaled by the transfer size if so indicated by `#imm`, where imm must be equal to $\log_2$(transfer_size). An assembler must accept Wm or Xm as an extended register offset, but Wm is preferred for disassembly.
- Generating an address lower than the value in the base register requires a negative signed immediate offset or a register offset holding a negative value.
- When stack alignment checking is enabled by system software and the base register is the SP, the current stack pointer must be initially quadword aligned, that is aligned to 16 bytes. Misalignment generates a Stack Alignment fault. The offset does not have to be a multiple of 16 bytes unless the specific load/store instruction requires this. SP cannot be used as a register offset.

C1.3.3.1 Address calculation

General-purpose arithmetic instructions can calculate the result of most addressing modes and write the address to a general-purpose register or, in most cases, to the current stack pointer.

Table C1-9 shows the arithmetic instructions that can compute addressing modes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C1-9 Arithmetic instructions to compute addressing modes

Addressing Form	Offset		
	Immediate	Register	Extended Register
Base register (no offset)	`MOV Xd|SP, base`	-	-

Base plus offset	`ADD Xd|SP, base, #imm`
	or
	`SUB Xd|SP, base, #imm`
		`ADD <Xd|SP>, base, Xm{,LSL#imm}`	`ADD <Xd|SP>, base, Wm,(S|U)XT(W|H|B|X) {#imm}`

Pre-indexed	-	-	-
Post-indexed	-	-	-
Literal (PC-relative)	`ADR Xd, label`	-	-

Note

- For the 64-bit base plus register offset form, the `UXTX` mnemonic is an alias for the `LSL` shift option, but `LSL` is preferred for disassembly. Similarly the `SXTX` extend/shift option is functionally equivalent to the `LSL` option, but the `LSL` option is preferred in source code.
- To calculate a base plus immediate offset the `ADD` instructions defined in Arithmetic (immediate) accept an unsigned 12-bit immediate offset, with an optional left shift by 12. This means that a single `ADD` instruction cannot support the full range of byte offsets available to a single register load/store with a scaled 12-bit immediate offset. For example, a quadword `LDR` effectively has a 16-bit byte offset. To calculate an address with a byte offset that requires more than 12 bits it is necessary to use two `ADD` instructions. The following example shows this:

```
ADD Xd, base, #(imm & 0xFFF)
ADD Xd, Xd, #(imm>>12), LSL #12
```

- To calculate a base plus extended register offset, the `ADD` instructions defined in Arithmetic (extended register) provide a superset of the addressing mode that also supports sign-extension or zero-extension of a byte or halfword value with any shift amount between 0 and 4, for example:

```
ADD  Xd, base, Wm, SXTW #3   // Xd = base + (SignExtend(Wm) LSL 3)
ADD  Xd, base, Wm, UXTH #4   // Xd = base + (ZeroExtend(Wm<15:0>) LSL 4)
```

- If the same extended register offset is used by more than one load/store instruction, then, depending on the implementation, it might be more efficient to calculate the extended and scaled intermediate result just once, and then reuse it as a simple register offset. The extend and scale calculation can be performed using the `SBFIZ` and `UBFIZ` bitfield instructions defined in Bitfield move, for example:

```
SBFIZ Xd, Xm, #3, #32   //Xd = "Wm, SXTW #3"
UBFIZ Xd, Xm, #4, #16   //Xd = "Wm, UXTH #4"
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.4 Instruction aliases

Some instructions have an associated *architecture alias* that is used for disassembly of the encoding when the associated conditions are met. Architecture alias instructions are included in the alphabetic lists of instruction types and clearly presented as an alias form in descriptions for the individual instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C2
About the A64 Instruction Descriptions

This chapter describes the *instruction descriptions* contained in A64 Base Instruction Descriptions, A64 Advanced SIMD and Floating-point Instruction Descriptions, SVE Instruction Descriptions, and SME Instruction Descriptions.

It contains the following sections:

- Understanding the A64 instruction descriptions.
- General information about the A64 instruction descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.1 Understanding the A64 instruction descriptions

Each instruction description in A64 Base Instruction Descriptions, A64 Advanced SIMD and Floating-point Instruction Descriptions, SVE Instruction Descriptions, and SME Instruction Descriptions has the following content:

1. A title.
2. An introduction to the instruction.
3. The instruction encoding or encodings.
4. Any alias conditions.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

The following sections describe each of these.

C2.1.1 The title

The title of an instruction description includes the base mnemonic for the instruction.

If different forms of an instruction use the same base mnemonic, each form has its own description. In this case, the title is the mnemonic followed by a short description of the instruction form in parentheses. This is most often used when an operand is an immediate value in one instruction form, but is a register in another form.

For example, in A64 Base Instruction Descriptions there are the following titles for different forms of the `ADD` instruction:

- ADD (extended register).
- ADD (immediate).
- ADD (shifted register).

C2.1.2 An introduction to the instruction

This briefly describes the function of the instruction. The introduction is not a complete description of the instruction, and it is not definitive. If there is any conflict between it and the more detailed information that follows it, the more detailed information takes priority.

C2.1.3 The instruction encoding or encodings

This shows the instruction encoding diagram, or if the instruction has more than one encoding, shows all of the encoding diagrams. Each diagram has a subheading.

For example, for load and store instructions, the subheadings might be:

- Post-index.
- Pre-index.
- Unsigned offset.

Each diagram numbers the bits from 31 to 0. The diagram for an instruction at address *A* shows, from left to right, the bytes at addresses *A*+3, *A*+2, *A*+1, and *A*.

There might be variants of an encoding, if the *assembler syntax prototype* differs depending on the value in one or more of the encoding fields. In this case, each variant has a subheading that describes the variant and shows the distinguishing field value or values in parentheses. For example, in A64 Base Instruction Descriptions there are the following subheadings for variants of the `ADC` instruction encoding:

- 32-bit variant (sf = 0).
- 64-bit variant (sf = 1).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The assembler syntax prototype for an encoding or variant of an encoding shows how to form a complete assembler source code instruction that assembles to the encoding. Unless otherwise stated, the prototype is also the preferred syntax for a disassembler to disassemble the encoding to. Disassemblers are permitted to omit optional symbols that represent the default value of a field or set of fields, to produce more readable disassembled code, provided that the output re-assembles to the same encoding.

Each encoding diagram, and its associated assembler syntax prototypes, is followed by encoding-specific pseudocode that translates the fields of that encoding into inputs for the encoding-independent pseudocode that describes the operation of the instruction. See Pseudocode describing how the instruction operates.

C2.1.4 Any alias conditions, if applicable

This is an optional part of an instruction description. If included, it describes the set of conditions for which an alternative mnemonic and its associated assembler syntax prototypes are preferred for disassembly by a disassembler. It includes a link to the alias instruction description that defines the alternative syntax. The alias syntax and the original syntax can be used interchangeably in the assembler source code.

Arm recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

C2.1.5 A list of the assembler symbols for the instruction

The *Assembler symbols* subsection of the instruction description contains a list of the symbols that the assembler syntax prototype or prototypes use, if any.

In assembler syntax prototypes, the following conventions are used:

`< >` Angle brackets. Any symbol enclosed by these is a name or a value that the user supplies. For each symbol, there is a description of what the symbol represents. The description usually also specifies which encoding field or fields encodes the symbol.

`{ }` Brace brackets. Any symbols enclosed by these are optional. For each optional symbol, there is a description of what the symbol represents and how its presence or absence is encoded.

In some assembler syntax prototypes, some brace brackets are mandatory, for example if they surround a register list. When the use of brace brackets is mandatory, they are separated from other syntax items by one or more spaces.

`#` This usually precedes a numeric constant. All uses of # are optional in A64 assembler source code. Arm recommends that disassemblers output the # where the assembler syntax prototype includes it.

`+/-` This indicates an optional + or – sign. If neither is coded, + is assumed.

Single spaces are used for clarity, to separate syntax items. Where a space is mandatory, the assembler syntax prototype shows two or more consecutive spaces.

Any characters not shown in this conventions list must be coded exactly as shown in the assembler syntax prototype. Apart from brace brackets, the characters shown are used as part of a meta-language to define the architectural assembler syntax for an instruction encoding or alias, but have no architecturally defined significance in the input to an assembler or in the output from a disassembler.

The following symbol conventions are used:

`<Xn>` The 64-bit name of a general-purpose register (X0-X30) or the zero register (XZR).

`<Wn>` The 32-bit name of a general-purpose register (W0-W30) or the zero register (WZR).

`<Xn|SP>` The 64-bit name of a general-purpose register (X0-X30) or the current stack pointer (SP).

`<Wn|WSP>` The 32-bit name of a general-purpose register (W0-W30) or the current stack pointer (WSP).

`<Bn>, <Hn>, <Sn>, <Dn>, <Qn>`

The 8, 16, 32, 64, or 128-bit name of a SIMD and floating-point register in a scalar context, as described in Register names.

`<Vn>` The name of a SIMD and floating-point register in a vector context, as described in Register names.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`<Zn>` The name of an SVE scalable vector register, as described in Treatment of SVE scalable vector registers.

`<Pn>` The name of an SVE scalable predicate register, as described in Vector predication.

If the description of a symbol specifies that the symbol is a register, the description might also specify that the range of permitted registers is extended or restricted. It also specifies any differences from the default rules for such fields.

For information about SME conventions, see ZA storage.

Note

Register names provides the A64 register names.

C2.1.6 Pseudocode describing how the instruction operates

The *Operation* subsection of the instruction description contains this pseudocode.

It is encoding-independent pseudocode that provides a precise description of what the instruction does.

Note

For a description of Arm pseudocode, see Arm Pseudocode Definition. This appendix also describes the execution model for an instruction.

C2.1.7 Notes, if applicable

If applicable, other notes about the instruction appear under additional subheadings.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.2 General information about the A64 instruction descriptions

This section provides general information about the A64 instruction descriptions. Some of this information also applies to System register descriptions, for example the terms defined in Fixed values in AArch64 instruction and System register descriptions apply to the AArch64 descriptions throughout this manual. The following subsections provide this information:

- Execution of instructions in Debug state.
- Fixed values in AArch64 instruction and System register descriptions.
- Modified immediate constants in A64 floating-point instructions.

C2.2.1 Execution of instructions in Debug state

In general, except for the instructions described in Debug state, the A64 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see Executing instructions in Debug state.

Note

For many instructions, execution is unchanged in Debug state. Executing instructions in Debug state identifies these instructions.

C2.2.2 Fixed values in AArch64 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

Note

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

RAZ Read-As-Zero. See Read-As-Zero (RAZ).

In diagrams, a RAZ bit can be shown as 0.

(0), RES0 Reserved, Should-Be-Zero (SBZ) or RES0.

In instruction encoding diagrams, and sometimes in other descriptions, `(0)` indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(0)(0)(0)` or as `(000)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. See the Glossary definition of RES0 for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,

The `(0)` and RES0 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RAO Read-As-One. See Read-As-One (RAO).

In diagrams, a RAO bit can be shown as 1.

(1), RES1 Reserved, Should-Be-One (SBO) or RES1.

In instruction encoding diagrams, and sometimes in other descriptions, `(1)` indicates an SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(1)(1)(1)` or as `(111)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. See the Glossary definition of *RES1* for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES1 fields,

The `(1)` and RES1 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

C2.2.3 Modified immediate constants in A64 floating-point instructions

Table C2-1 shows the immediate constants available in `FMOV` (scalar, immediate) and `FMOV` (vector, immediate) floating-point instructions.

Table C2-1 A64 Floating-point modified immediate constants

Data type	immediate	Constant[a]
`F16`	abcdefgh	`aBbbcdef gh000000`
`F32`	abcdefgh	`aBbbbbbc defgh000 00000000 00000000`
`F64`	abcdefgh	`aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000`

a. In this column, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

The immediate value shown in the table is either:

- The value of the `imm8` field for an `FMOV` (scalar, immediate) instruction, see FMOV (scalar, immediate).
- The value obtained by concatenating the `a:b:c:d:e:f:g:h` fields for an `FMOV` (vector, immediate) instruction, see FMOV (vector, immediate).

Table C2-2 shows the floating-point constant values encoded in the `b:c:d:e:f:g:h` fields of the `FMOV` (vector, immediate) instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C2-2 Floating-point constant values

efgh	bcd							
	000	001	010	011	100	101	110	111
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

C2.2.3.1 Operation of modified immediate constants, floating-point instructions

For an A64 floating-point instruction that uses a modified immediate constant, the operation described by the `VFPExpandImm()` pseudocode function returns the value of the immediate constant.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C3
A64 Instruction Set Overview

This chapter provides an overview of the A64 instruction set. It contains the following sections:

- Branches, Exception generating, and System instructions.
- Loads and stores.
- Loads and stores - SVE, SVE2.
- Loads and stores - SME, SME2, SVE2p1.
- Data processing - immediate.
- Data processing - MTE.
- Data processing - register.
- Data processing - SIMD and floating-point.
- Data processing - SVE.
- Data processing - SVE2.
- Data processing - SME, SME2.

For a structured breakdown of instruction groups by encoding, see A64 Instruction Set Encoding.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.1 Branches, Exception generating, and System instructions

This section describes the branch, exception generating, and System instructions. It contains the following subsections:

- Conditional branch.
- Unconditional branch (immediate).
- Unconditional branch (register).
- Direct and indirect branch.
- Exception generation and return.
- System register instructions.
- System instructions.
- Hint instructions.
- Barriers and CLREX instructions.
- Pointer authentication instructions.

Note

Software must:

- Use only `BLR` or `BL` to perform a nested subroutine call when that subroutine is expected to return to the immediately following instruction, that is, the instruction with the address of the `BLR` or `BL` instruction incremented by four.
- Use only `RET` to perform a subroutine return, when that subroutine is expected to have been entered by a `BL` or `BLR` instruction.
- Use only `B`, `BR`, or the instructions listed in Table C3-1 to perform a control transfer that is not a subroutine call or subroutine return described in this *Note*.

C3.1.1 Conditional branch

Conditional branches change the flow of execution depending on the following:

- The current state of the Condition flags.
- The result of comparison between 0 and a value or a bit in a general purpose register.
- The result of comparison between values in two general purpose registers.
- The result of comparison between an immediate value and a value in a general purpose register.

See Table C1-1 for a list of the Condition codes that can be used for `cond`.

Table C3-1 shows the Conditional branch instructions.

Table C3-1 Conditional branch instructions

Mnemonic	Instruction	Branch offset range from the PC	See
`B.cond`	Branch conditionally	±1MB	B.cond
`BC.cond`	Branch Consistent conditionally	±1MB	BC.cond
`CBEQ`	Compare equal to immediate and branch	±1KB	CB<cc> (immediate)
	Compare equal to register and branch		CB<cc> (register)
`CBGE`	Compare signed greater than or equal to immediate and branch	±1KB	CBGE (immediate)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	Branch offset range from the PC	See
	Compare signed greater than or equal to register and branch		CB<cc> (register)
CBGT	Compare signed greater than immediate and branch	±1KB	CB<cc> (immediate)
	Compare signed greater than register and branch		CB<cc> (register)
CBHI	Compare unsigned higher than immediate and branch	±1KB	CB<cc> (immediate)
	Compare unsigned higher than register and branch		CB<cc> (register)
CBHS	Compare unsigned higher than or same as immediate and branch	±1KB	CBHS (immediate)
	Compare unsigned higher than or same as register and branch		CB<cc> (register)
CBLE	Compare signed less than or equal to immediate and branch	±1KB	CBLE (immediate)
	Compare signed less than or equal to register and branch		CBLE (register)
CBLO	Compare unsigned lower than immediate and branch	±1KB	CB<cc> (immediate)
	Compare unsigned lower than register and branch		CBLO (register)
CBLS	Compare unsigned lower than or same as immediate and branch	±1KB	CBLS (immediate)
	Compare unsigned lower than or same as register and branch		CBLS (register)
CBLT	Compare signed less than immediate and branch	±1KB	CB<cc> (immediate)
	Compare signed less than register and branch		CBLT (register)
CBNE	Compare not equal to immediate and branch	±1KB	CB<cc> (immediate)
	Compare not equal to register and branch		CB<cc> (register)
CBBEQ	Compare equal to bytes and branch	±1KB	CBB<cc>
CBBGE	Compare signed greater than or equal to bytes and branch		
CBBGT	Compare signed greater than bytes and branch		
CBBHI	Compare unsigned higher than bytes and branch		
CBBHS	Compare unsigned higher than or same as bytes and branch		
CBBLE	Compare signed less than or equal to bytes and branch	±1KB	CBBLE
CBBLO	Compare unsigned lower than bytes and branch	±1KB	CBBLO
CBBLS	Compare unsigned lower than or same as bytes and branch	±1KB	CBBLS
CBBLT	Compare signed less than bytes and branch	±1KB	CBBLT
CBBNE	Compare not equal to bytes and branch	±1KB	CBB<cc>
CBHEQ	Compare equal to halfwords and branch	±1KB	CBH<cc>
CBHGE	Compare signed greater than or equal to halfwords and branch		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	Branch offset range from the PC	See
CBHGT	Compare signed greater than halfwords and branch		
CBHHI	Compare unsigned higher than halfwords and branch		
CBHHS	Compare unsigned higher than or same as halfwords and branch		
CBHLE	Compare signed less than or equal halfwords and branch	±1KB	CBHLE
CBHLO	Compare unsigned lower than halfwords and branch	±1KB	CBHLO
CBHLS	Compare unsigned lower than or same as halfwords and branch	±1KB	CBHLS
CBHLT	Compare signed less than halfwords and branch	±1KB	CBHLT
CBHNE	Compare not equal to halfwords and branch	±1KB	CBH<cc>
CBNZ	Compare and branch if nonzero	±1MB	CBNZ
CBZ	Compare and branch if zero	±1MB	CBZ
TBNZ	Test bit and branch if nonzero	±32KB	TBNZ
TBZ	Test bit and branch if zero	±32KB	TBZ

C3.1.2 Unconditional branch (immediate)

Unconditional branch (immediate) instructions change the flow of execution unconditionally by adding an immediate offset with a range of ±128MB to the value of the program counter that fetched the instruction. The BL instruction also writes the address of the sequentially following instruction to general-purpose register, X30.

Table C3-2 shows the Unconditional branch instructions with an immediate branch offset.

Table C3-2 Unconditional branch instructions (immediate)

Mnemonic	Instruction	Immediate branch offset range from the PC	See
B	Branch unconditionally	±128MB	B
BL	Branch with link	±128MB	BL

C3.1.3 Unconditional branch (register)

Unconditional branch (register) instructions change the flow of execution unconditionally by setting the program counter to the value in a general-purpose register. The BLR instruction also writes the address of the sequentially following instruction to general-purpose register X30. The RET instruction behaves identically to BR, but provides an additional hint to the PE that this is a return from a subroutine. Table C3-3 shows Unconditional branch instructions that jump directly to an address held in a general-purpose register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-3 Unconditional branch instructions (register)

Mnemonic	Instruction	See
BLR	Branch with link to register	BLR
BR	Branch to register	BR
RET	Return from subroutine	RET

C3.1.4 Direct and indirect branch

Table C3-4 and Table C3-5 shows the Direct branch and Indirect branch instructions.

Table C3-4 Direct branch instructions

Mnemonic	Instruction	See
B	Unconditional branch	B
B.cond	Conditional branch	B.cond
BC.cond	Branch consistent conditionally	BC.cond
BL	Branch with link	BL
CB<cc>	Compare register with immediate and branch	CB<cc> (immediate)
	Compare registers and branch	CB<cc> (register)
CBB<cc>	Compare bytes and branch	CBB<cc>
CBH<cc>	Compare halfwords and branch	CBH<cc>
CBZ	Compare and branch on zero	CBZ
CBNZ	Compare and branch on nonzero	CBNZ
TBZ	Test bit and branch if zero	TBZ
TBNZ	Test bit and branch if nonzero	TBZ

Table C3-5 Indirect branch instructions

Mnemonic	Instruction	See
BLR	Branch with link to register	BLR
BLRAA	Branch with link to register, with pointer authentication	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRAAZ		
BLRAB		
BLRABZ		
BR	Branch to register	BR
BRAA	Branch to register, with pointer authentication	BRAA, BRAAZ, BRAB, BRABZ
BRAAZ		
BRAB		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
BRABZ		
ERET	Exception return	ERET
ERETAA	Exception return, with pointer authentication	ERETAA, ERETAB
ERETAB		
RET	Return from subroutine	RET
RETAA	Return from subroutine, with pointer authentication	RETAA, RETAB
RETAB		
RETAASPPC	Return from subroutine, with enhanced pointer authentication using an immediate offset	RETAASPPC, RETABSPPC
RETABSPPC		
RETAASPPCR	Return from subroutine, with enhanced pointer authentication using a register	RETAASPPCR, RETABSPPCR
RETABSPPCR		

C3.1.5 Exception generation and return

This section describes the following exceptions:

- Exception generating.
- Exception return.
- Debug state.

C3.1.5.1 Exception generating

Table C3-6 shows the Exception generating instructions.

Table C3-6 Exception generating instructions

Mnemonic	Instruction	See
BRK	Breakpoint Instruction	BRK
HLT	Halt Instruction	HLT
HVC	Generate exception targeting Exception level 2	HVC
SMC	Generate exception targeting Exception level 3	SMC
SVC	Generate exception targeting Exception level 1	SVC

C3.1.5.2 Exception return

Table C3-7 shows the Exception return instructions.

Table C3-7 Exception return instructions

Mnemonic	Instruction	See
ERET	Exception return using current ELR and SPSR	ERET

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
ERETAA, ERETAB	Exception return, with pointer authentication	ERETAA, ERETAB

C3.1.5.3 Debug state

Table C3-8 shows the Debug state instructions.

Table C3-8 Debug state instructions

Mnemonic	Instruction	See
DCPS1	Debug switch to Exception level 1	DCPS1
DCPS2	Debug switch to Exception level 2	DCPS2
DCPS3	Debug switch to Exception level 3	DCPS3
DRPS	Debug restore PE state	DRPS

C3.1.6 System register instructions

For detailed information about the System register instructions, see The A64 System Instruction Class. Table C3-9 shows the System register instructions.

If FEAT_SYSREG128 is implemented, the following instructions are added that allow the PE to move values between a 128-bit System register and two adjacent 64-bit general-purpose registers:

- MRRS.
- MSRR.

Table C3-9 System register instructions

Mnemonic	Instruction	See
MRS	Move System register to general-purpose register	MRS
MSR	Move general-purpose register to System register	MSR (register)
	Move immediate to PE state field	MSR (immediate)
MRRS	Move 128-bit System register to two adjacent 64-bit general-purpose registers	MRRS
MSRR	Move two adjacent 64-bit general-purpose registers to 128-bit System register	MSRR

C3.1.7 System instructions

For detailed information about the System instructions, see The A64 System Instruction Class.

If FEAT_SYSINSTR128 is implemented, the 128-bit System instruction, SYSP, is supported.

Table C3-10 shows the System instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-10 System instructions

Mnemonic	Instruction	See
SYS	System instruction	SYS
SYSL	System instruction with result	SYSL
SYSP	128-bit System instruction	SYSP
IC	Instruction cache maintenance	IC and Table C5-1
DC	Data cache maintenance	DC and Table C5-1
AT	Address translation	AT and Table C5-3
TLBI	TLB Invalidate	TLBI and Table C5-4

C3.1.8 Hint instructions

Table C3-11 shows the Hint instructions.

Table C3-11 Hint instructions

Mnemonic	Instruction	See
NOP	No operation	NOP
YIELD	Yield hint	YIELD
WFE	Wait for event	WFE
WFET	Wait for event with timeout	WFET
WFI	Wait for interrupt	WFI
WFIT	Wait for interrupt with timeout	WFIT
SEV	Send event	SEV
SEVL	Send event local	SEVL
HINT	Unallocated hint	HINT
DGH	Data Gathering Hint	DGH
CLRBHB	Clear Branch History	CLRBHB
CHKFEAT	Check Feature	CHKFEAT
STSHH	Store Shared Hint	STSHH

C3.1.9 Barriers and CLREX instructions

Table C3-12 shows the barrier and CLREX instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-12 Barriers and CLREX instructions

Mnemonic	Instruction	See
CLREX	Clear Exclusives monitor	CLREX
DMB	Data memory barrier	DMB
DSB	Data synchronization barrier	DSB
ISB	Instruction synchronization barrier	ISB

For more information about DSB, DMB, and ISB, see Memory barriers.

Table C3-13 shows the speculation and synchronization barriers. If these instructions are not implemented, then these instructions execute as a NOP.

Table C3-13 Speculation and synchronization barriers

Mnemonic	Instruction	See
CSDB	Consumption of Speculative Data Barrier	CSDB
ESB	Error synchronization barrier	ESB
PSB	Profiling synchronization barrier	PSB
PSSBB	Physical Speculative Store Bypass Barrier	PSSBB
SB	Speculation Barrier	SB
SSBB	Speculative Store Bypass Barrier	SSBB
TSB	Trace Synchronization Barrier	TSB
GCSB	Guarded Control Stack Barrier	GCSB

For more information about:

- CSDB, PSSBB, SB, SSBB, TSB, see Memory barriers.
- ESB, see Error synchronization event.
- PSB, see The Statistical Profiling Extension.
- GCSB, see Guarded Control Stack data accesses.

C3.1.10 Pointer authentication instructions

FEAT_PAuth adds support for pointer authentication, see Pointer authentication. This functionality includes the A64 instructions described in this section. These instructions fall into two groups, see:

- Basic pointer authentication instructions.
- Combined instructions that include pointer authentication.

C3.1.10.1 Basic pointer authentication instructions

Each of these instructions only performs an operation that supports pointer authentication.

Table C3-14 shows the instructions that add a *Pointer Authentication Code* (PAC) to the address in a register:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-14 Instructions that add a PAC

Mnemonic	Instruction	See
PACIASP	Add PAC to instruction address using APIAKey_EL1 and SP	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACIAZ	Add PAC to instruction address using APIAKey_EL1 and zero	
PACIA1716	Add PAC to instruction address X17 using APIAKey_EL1 and X16	
PACIA171615	Add PAC to instruction address X17 using APIAKey_EL1, X16, X15	PACIA171615
PACIASPPC	Add PAC to instruction address LR using APIAKey_EL1, SP, PC	PACIASPPC
PACNBIASPPC	Add PAC to instruction address LR using APIAKey_EL1, SP, PC	PACNBIASPPC
PACIBSP	Add PAC to instruction address using APIBKey_EL1 and SP	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACIBZ	Add PAC to instruction address using APIBKey_EL1 and zero	
PACIB1716	Add PAC to instruction address X17 using APIBKey_EL1 and X16	
PACIB171615	Add PAC to instruction address X17 using APIBKey_EL1, X16, X15	PACIB171615
PACIBSPPC	Add PAC to instruction address LR using APIBKey_EL1, SP, PC	PACIBSPPC
PACNBIBSPPC	Add PAC to instruction address LR using APIBKey_EL1, SP, PC	PACNBIBSPPC
PACIA	Add PAC to instruction address using APIAKey_EL1, registers	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACDA	Add PAC to data address using APDAKey_EL1, registers	PACDA, PACDZA
PACIB	Add PAC to instruction address using APIBKey_EL1, registers	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACDB	Add PAC to data address using APDBKey_EL1, registers	PACDB, PACDZB
PACIZA	Add PAC to instruction address using APIAKey_EL1, register and zero	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACDZA	Add PAC to data address using APDAKey_EL1, register and zero	PACDA, PACDZA
PACIZB	Add PAC to instruction address using APIBKey_EL1, register and zero	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACDZB	Add PAC to data address using APDBKey_EL1, register and zero	PACDB, PACDZB
PACGA	Add generic PAC using APGAKey_EL1, registers	PACGA

Table C3-15 shows the instructions that authenticate a PAC in a register:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-15 Instructions that authenticate a PAC

Mnemonic	Instruction	See
AUTIASP	Authenticate PAC for instruction address using APIAKey_EL1 and SP	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTIAZ	Authenticate PAC for instruction address using APIAKey_EL1 and zero	
AUTIA1716	Authenticate PAC for instruction address X17 using APIAKey_EL1 and X16	
AUTIA171615	Authenticate PAC for instruction address X17 using APIAKey_EL1, X16, and X15	AUTIA171615
AUTIASPPC	Authenticate PAC for instruction address LR using APIAKey_EL1, SP, and PC with an immediate offset	AUTIASPPC
AUTIASPPCR	Authenticate PAC for instruction address LR using APIAKey_EL1, SP, and a register	AUTIASPPCR
AUTIBSP	Authenticate PAC for instruction address using APIBKey_EL1 and SP	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTIBZ	Authenticate PAC for instruction address using APIBKey_EL1 and zero	
AUTIB1716	Authenticate PAC for instruction address X17 using APIBKey_EL1 and X16	
AUTIB171615	Authenticate PAC for instruction address X17 using APIBKey_EL1, X16, X15	AUTIB171615
AUTIBSPPC	Authenticate PAC for instruction address LR using APIBKey_EL1, SP, and PC with an immediate offset	AUTIBSPPC
AUTIBSPPCR	Authenticate PAC for instruction address LR using APIBKey_EL1, SP, and a register	AUTIBSPPCR
AUTIA	Authenticate PAC for instruction address using APIAKey_EL1, registers	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDA	Authenticate PAC for data address using APDAKey_EL1, registers	AUTDA, AUTDZA
AUTIB	Authenticate PAC for instruction address using APIBKey_EL1, registers	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDB	Authenticate PAC for data address using APDBKey_EL1, registers	AUTDB, AUTDZB
AUTIZA	Authenticate PAC for instruction address using APIAKey_EL1, register and zero	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDZA	Authenticate PAC for data address using APDAKey_EL1, register and zero	AUTDA, AUTDZA
AUTIZB	Authenticate PAC for instruction address using APIBKey_EL1, register and zero	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDZB	Authenticate PAC for data address using APDBKey_EL1, register and zero	AUTDB, AUTDZB

Table C3-15 shows the instructions that authenticate a PAC in a register:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-16 Instructions that authenticate a PAC

Mnemonic	Instruction	See
AUTIASP	Authenticate PAC for instruction address using APIAKey_EL1 and SP	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTIAZ	Authenticate PAC for instruction address using APIAKey_EL1 and zero	
AUTIA1716	Authenticate PAC for instruction address X17 using APIAKey_EL1 and X16	
AUTIBSP	Authenticate PAC for instruction address using APIBKey_EL1 and SP	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTIBZ	Authenticate PAC for instruction address using APIBKey_EL1 and zero	
AUTIB1716	Authenticate PAC for instruction address X17 using APIBKey_EL1 and X16	
AUTIA	Authenticate PAC for instruction address using APIAKey_EL1, registers	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDA	Authenticate PAC for data address using APDAKey_EL1, registers	AUTDA, AUTDZA
AUTIB	Authenticate PAC for instruction address using APIBKey_EL1, registers	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDB	Authenticate PAC for data address using APDBKey_EL1, registers	AUTDB, AUTDZB
AUTIZA	Authenticate PAC for instruction address using APIAKey_EL1, register and zero	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDZA	Authenticate PAC for data address using APDAKey_EL1, register and zero	AUTDA, AUTDZA
AUTIZB	Authenticate PAC for instruction address using APIBKey_EL1, register and zero	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDZB	Authenticate PAC for data address using APDBKey_EL1, register and zero	AUTDB, AUTDZB

Table C3-17 shows the instructions that strip a PAC from a register, without performing any authentication:

Table C3-17 Instructions that strip a PAC

Mnemonic	Instruction	See
XPACLRI	Strip instruction address PAC from LR	XPACD, XPACI, XPACLRI
XPACI	Strip instruction address PAC, register	
XPACD	Strip data address PAC, register	

Table C3-18 shows the instructions that set PSTATE.PACM to 1:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-18 Instructions that set PSTATE.PACM to 1

Mnemonic	Instruction	See
PACM	Pointer authentication modifier	PACM

C3.1.10.2 Combined instructions that include pointer authentication

Each of these instructions combines a pointer authentication with another operation that uses the authenticated pointer. Table C3-19 shows these instructions:

Table C3-19 Combined pointer authentication instructions

Mnemonic	Instruction	See
RETAA	Authenticate PAC for LR using APIAKey_EL1 and SP, and return	RETAA, RETAB
RETAB	Authenticate PAC for LR using APIBKey_EL1 and SP, and return	
RETAASPPC	Authenticate PAC for LR using APIAKey_EL1, SP, and PC with an immediate offset, and return	RETAASPPC, RETABSPPC
RETABSPPC	Authenticate PAC for LR using APIBKey_EL1, SP, and PC with an immediate offset, and return	
RETAASPPCR	Authenticate PAC for LR using APIAKey_EL1, SP, and a register, and return	RETAASPPCR, RETABSPPCR
RETABSPPCR	Authenticate PAC for LR using APIBKey_EL1, SP, and a register, and return	
BRAA	Authenticate PAC using APIAKey_EL1 (registers), and branch	BRAA, BRAAZ, BRAB, BRABZ
BRAB	Authenticate PAC using APIBKey_EL1 (registers), and branch	
BLRAA	Authenticate PAC using APIAKey_EL1 (registers), and branch with link	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRAB	Authenticate PAC using APIBKey_EL1 (registers), and branch with link	
BRAAZ	Authenticate PAC using APIAKey_EL1 (register and zero), and branch	BRAA, BRAAZ, BRAB, BRABZ
BRABZ	Authenticate PAC using APIBKey_EL1 (register and zero), and branch	
BLRAAZ	Authenticate PAC using APIAKey_EL1 (register and zero), and branch with link	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRABZ	Authenticate PAC using APIBKey_EL1 (register and zero), and branch with link	
ERETAA	Authenticate PAC for ELR using APIAKey_EL1 and SP, and exception return	ERETAA, ERETAB
ERETAB	Authenticate PAC for ELR using APIBKey_EL1 and SP, and exception return	
LDRAA	Authenticate PAC for data address using APDAKey_EL1 (register and zero) and Load	LDRAA, LDRAB
LDRAB	Authenticate PAC for data address using APDBKey_EL1 (register and zero) and Load	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.1.11 Checked Pointer Arithmetic instructions

FEAT_CPA adds support for Checked Pointer Arithmetic, see Checked Pointer Arithmetic.

If FEAT_CPA is implemented, the instructions shown in Table C3-20 are implemented.

Table C3-20 Checked Pointer Arithmetic instructions

Mnemonic	Instruction	See
ADDPT	Scalar add checked pointer	ADDPT
	SVE add checked pointer (predicated)	ADDPT
	SVE add checked pointer (unpredicated)	ADDPT
SUBPT	Scalar subtract checked pointer	SUBPT
	SVE subtract checked pointer (predicated)	SUBPT
	SVE subtract checked pointer (unpredicated)	SUBPT
MADDPT	Scalar multiply-add checked pointer	MADDPT
MADPT	SVE multiply-add checked pointer vectors, writing multiplicand	MADPT
MLAPT	SVE Multiply-add checked pointer vectors, writing addend	MLAPT
MSUBPT	Scalar multiply-subtract checked pointer	MSUBPT

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2 Loads and stores

This section describes the load/store instructions. It contains the following subsections:

- Load/store register.
- Load/store register (unscaled offset).
- Load/store pair.
- Load/store pair unprivileged.
- Load/store non-temporal pair.
- Load/store non-temporal pair unprivileged.
- Load/store unprivileged.
- Load-Exclusive/Store-Exclusive.
- Load-Exclusive/Store-Exclusive unprivileged.
- Load-Acquire/Store-Release.
- Load-Acquire/Store-Release unprivileged.
- LoadLOAcquire/StoreLORelease.
- Load/store scalar SIMD and floating-point.
- Load/store Advanced SIMD.
- Prefetch memory.
- Atomic instructions.
- Memory Tagging instructions.
- Memory Copy and Memory Set instructions.

The requirements for the alignment of data memory accesses are strict. For more information, see Alignment of data accesses.

The additional control bits SCTLR_ELx.SA and SCTLR_EL1.SA0 control whether the stack pointer must be quadword aligned when used as a base register. See SP alignment checking. Using a misaligned stack pointer generates an SP alignment fault exception.

Note

In some cases, load/store instructions can lead to CONSTRAINED UNPREDICTABLE behavior. See AArch64 CONSTRAINED UNPREDICTABLE behaviors.

C3.2.1 Load/store register

The load/store register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset, optionally scaled.
- Base plus a 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

See also Load/store addressing modes.

If a Load instruction specifies writeback and the register being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs the load using the specified addressing mode and the base register becomes UNKNOWN. In addition, if an exception occurs during the execution of such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store instruction performs a writeback and the register that is stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the store to the designated register using the specified addressing mode, but the value stored is UNKNOWN.

Table C3-21 shows the load/store register instructions.

Table C3-21 Load/store register instructions

Mnemonic	Instruction	See
LDR	Load register (register offset)	LDR (register)
	Load register (immediate offset)	LDR (immediate)
	Load register (PC-relative literal)	LDR (literal)
LDRB	Load byte (register offset)	LDRB (register)
	Load byte (immediate offset)	LDRB (immediate)
LDRSB	Load signed byte (register offset)	LDRSB (register)
	Load signed byte (immediate offset)	LDRSB (immediate)
LDRH	Load halfword (register offset)	LDRH (register)
	Load halfword (immediate offset)	LDRH (immediate)
LDRSH	Load signed halfword (register offset)	LDRSH (register)
	Load signed halfword (immediate offset)	LDRSH (immediate)
LDRSW	Load signed word (register offset)	LDRSW (register)
	Load signed word (immediate offset)	LDRSW (immediate)
	Load signed word (PC-relative literal)	LDRSW (literal)
STR	Store register (register offset)	STR (register)
	Store register (immediate offset)	STR (immediate)
STRB	Store byte (register offset)	STRB (register)
	Store byte (immediate offset)	STRB (immediate)
STRH	Store halfword (register offset)	STRH (register)
	Store halfword (immediate offset)	STRH (immediate)

C3.2.2 Load/store register (unscaled offset)

The load/store register instructions with an unscaled offset support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See Load/store addressing modes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The load/store register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store register instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset, because that can represent some offset values in the same range.

The ambiguous immediate offsets are byte offsets that are both:

- In the range 0-255, inclusive.
- Naturally aligned to the access size.

Other byte offsets in the range -256 to 255 inclusive are unambiguous. An assembler program translating a load/store instruction, for example `LDR`, is required to encode an unambiguous offset using the unscaled 9-bit offset form, and to encode an ambiguous offset using the scaled 12-bit offset form. A programmer might force the generation of the unscaled 9-bit form by using one of the mnemonics in Table C3-22. Arm recommends that a disassembler outputs all unscaled 9-bit offset forms using one of these mnemonics, but unambiguous offsets can be output using a load/store single register mnemonic, for example, `LDR`.

Table C3-22 shows the load/store register instructions with an unscaled offset.

Table C3-22 Load/store register (unscaled offset) instructions

Mnemonic	Instruction	See
`LDUR`	Load register (unscaled offset)	LDUR
`LDURB`	Load byte (unscaled offset)	LDURB
`LDURSB`	Load signed byte (unscaled offset)	LDURSB
`LDURH`	Load halfword (unscaled offset)	LDURH
`LDURSH`	Load signed halfword (unscaled offset)	LDURSH
`LDURSW`	Load signed word (unscaled offset)	LDURSW
`STUR`	Store register (unscaled offset)	STUR
`STURB`	Store byte (unscaled offset)	STURB
`STURH`	Store halfword (unscaled offset)	STURH

C3.2.3 Load/store pair

The load/store pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

If a Load Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

If a Load Pair instruction specifies writeback and one of the registers being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the base register becomes UNKNOWN. In addition, if an exception occurs during the instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store Pair instruction performs a writeback and one of the registers being stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the stores of the registers indicated by the specified addressing mode, but the value stored for the base register is UNKNOWN.

Table C3-23 shows the load/store pair instructions.

Table C3-23 Load/store pair instructions

Mnemonic	Instruction	See
LDP	Load Pair	LDP
LDPSW	Load Pair signed words	LDPSW
STP	Store Pair	STP

C3.2.4 Load/store pair unprivileged

The Load unprivileged pair and Store unprivileged pair instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load/store unprivileged pair instructions support the following address modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See Load/store addressing modes.

Table C3-24 Load/store unprivileged pair instructions

Mnemonic	Instruction	See
LDTP	Load unprivileged Pair	LDTP
STTP	Store unprivileged Pair	STTP

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.5 Load/store non-temporal pair

The load/store non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See Load/store addressing modes.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be prefetched and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load Non-temporal Pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-Temporal Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-25 shows the load/store non-temporal pair instructions.

Table C3-25 Load/store non-temporal pair instructions

Mnemonic	Instruction	See
LDNP	Load Non-temporal Pair	LDNP
STNP	Store Non-temporal Pair	STNP

C3.2.6 Load/store non-temporal pair unprivileged

The load/store non-temporal pair unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These intructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load/store non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

Table C3-26 Load/store non-temporal pair instructions

Mnemonic	Instruction	See
LDTNP	Load Non-temporal Pair	LDTNP
STTNP	Store Non-temporal Pair	STTNP

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.7 Load/store unprivileged

The load/store unprivileged instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See Load/store addressing modes.

The access permissions that apply to accesses made at EL0 apply to the memory accesses made by a load/store unprivileged instruction that is executed either:

- At EL1 when the Effective value of PSTATE.UAO is 0.
- At EL2 when both the Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and the Effective value of PSTATE.UAO is 0.

Otherwise, memory accesses made by a load/store unprivileged instruction are subject to the access permissions that apply to the Exception level at which the instruction is executed. These are the permissions that apply to the corresponding load/store register instruction, see Load/store register.

Note

This means that when the value of PSTATE.UAO is 1 the access permissions for a load/store unprivileged instruction are always the same as those for the corresponding load/store register instruction.

Table C3-27 shows the load/store unprivileged instructions.

Table C3-27 Load-Store unprivileged instructions

Mnemonic	Instruction	See
`LDTR`	Load unprivileged register	LDTR
`LDTRB`	Load unprivileged byte	LDTRB
`LDTRSB`	Load unprivileged signed byte	LDTRSB
`LDTRH`	Load unprivileged halfword	LDTRH
`LDTRSH`	Load unprivileged signed halfword	LDTRSH
`LDTRSW`	Load unprivileged signed word	LDTRSW
`STTR`	Store unprivileged register	STTR
`STTRB`	Store unprivileged byte	STTRB
`STTRH`	Store unprivileged halfword	STTRH

C3.2.8 Single-copy atomic 64-byte load/store

If FEAT_LS64 is implemented, the following instructions are implemented:

- LD64B.
- ST64B.

If FEAT_LS64_V is implemented, the following instructions are implemented:

- LD64B.
- ST64B.
- ST64BV.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_LS64_ACCDATA is implemented, the following instructions are implemented:

- LD64B.
- ST64B.
- ST64BV.
- ST64BV0.

LD64B and ST64B can only target Device or Normal Non-cacheable memory unless FEAT_LS64WB is implemented.

The single-copy atomic 64-byte load/store instructions support one addressing mode:

- Base register only.

See also Load/store addressing modes.

The memory location accessed by the instructions is required to be aligned on a 64-byte boundary, otherwise an Alignment fault occurs.

If an ST64BV, ST64BV0, or if FEAT_LS64WB is not implemented an LD64B or ST64B, attempts to access a memory type that is not one of the following, then a Data Abort with DFSC code `0b110101` is generated:

- Normal Inner Non-cacheable, Outer Non-cacheable.
- Device-GRE.
- Device-nGRE.
- Device-nGnRE.
- Device-nGnRnE.

It is IMPLEMENTATION DEFINED which of the following approaches is used to provide this check:

- The check is performed at each enabled stage of translation, and the fault is reported for the first stage of translation that provides an inappropriate memory type. In this case, the value of the HCR_EL2.DC bit does not cause accesses generated by the instructions to generate a stage 1 Data abort,
- The check is performed against the resulting memory type after all enabled stages of translation. In this case the fault is reported at the final enabled stage of translation.
- The check is performed against the resulting memory type after all enabled stages of translation. In this case the fault is reported for the first stage of translation that provided an inappropriate memory type. The value of the HCR_EL2.DC bit does not cause accesses generated by the instructions to generate a stage 1 Data abort.

If FEAT_LS64WB is implemented, then LD64B and ST64B are also supported and 64-byte single-copy atomic for Conventional memory that is Inner and Outer Write-Back Cacheable, Shareable. In this case, if an LD64B or ST64B attempts to access:

- A memory region that is Inner and Outer Write-Back Cacheable, Shareable, but the target memory location is not in Conventional memory, then it is IMPLEMENTATION DEFINED whether:
  - A stage 1 Data Abort with DFSC code `0b110101` is generated.
  - The access succeeds but is not guaranteed to be single-copy atomic above the byte level.
- A cacheable type of memory other than Inner and Outer Write-Back Cacheable, Shareable, then it is IMPLEMENTATION DEFINED whether:
  - A stage 1 Data Abort with DFSC code `0b110101` is generated.
  - The access succeeds as a 64-byte single-copy atomic operation.

For behavior when FEAT_LS64WB is implemented and an LD64B or ST64B targets Tagged memory that is Inner and Outer Write-Back Cacheable, Shareable, see Tag Check Faults.

If the target memory location does not support the LD64B or ST64B instructions, then one of the following behaviors occurs:

- A stage 1 Data Abort with DFSC code of `0b110101`, is generated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs the memory accesses, but the accesses are not single-copy atomic above the byte level.

If the target memory location does not support the ST64BV or ST64BV0 instructions, then the register specified by <Xs> is set to `0xFFFF_FFFF_FFFF_FFFF`.

Regardless of the memory type:

- The memory access generated by an ST64BV or ST64BV0 instruction is not merged with any accesses.
- The memory access generated by an ST64B instruction is not merged with any accesses generated by store instructions appearing in program order after the instruction.

Table C3-28 shows the single-copy atomic 64-byte load/store instructions.

Table C3-28 Single-copy atomic 64-byte load/store instructions

Mnemonic	Instruction	See
`LD64B`	Single-copy atomic 64-byte load	LD64B
`ST64B`	Single-copy atomic 64-byte store without status result	ST64B

Table C3-29 shows the single-copy atomic 64-byte store with status result.

Table C3-29 Single-copy atomic 64-byte store with status result instructions

Mnemonic	Instruction	See
`ST64BV`	Single-copy atomic 64-byte store with status result	ST64BV

Table C3-30 shows the single-copy atomic 64-byte EL0 store with status result.

Table C3-30 Single-copy atomic 64-byte EL0 store with status result instructions

Mnemonic	Instruction	See
`ST64BV0`	Single-copy atomic 64-byte EL0 store with status result	ST64BV0

C3.2.9 Load-Exclusive/Store-Exclusive

The Load-Exclusive/Store-Exclusive instructions support only one addressing mode:

- Base register with no offset.

See Load/store addressing modes.

The Load-Exclusive instructions mark the physical address being accessed as an exclusive access. This exclusive access mark is checked by the Store-Exclusive instruction, permitting the construction of atomic read-modify-write operations on shared memory variables, semaphores, mutexes, and spinlocks. See Synchronization and semaphores.

If FEAT_LSE2 is not implemented then:.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Load-Exclusive/Store-Exclusive instructions other than Load-Exclusive pair and Store-Exclusive pair require natural alignment, and an unaligned address generates an Alignment fault.
- Memory accesses generated by Load-Exclusive pair or Store-Exclusive pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault.

  For more information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

When a Store-Exclusive pair succeeds, it causes a single-copy atomic update of the entire memory location being stored to.

Table C3-31 shows the Load-Exclusive/Store-Exclusive instructions.

Table C3-31 Load-Exclusive/Store-Exclusive instructions

Mnemonic	Instruction	See
`LDXR`	Load Exclusive register	LDXR
`LDXRB`	Load Exclusive byte	LDXRB
`LDXRH`	Load Exclusive halfword	LDXRH
`LDXP`	Load Exclusive pair	LDXP
`STXR`	Store Exclusive register	STXR
`STXRB`	Store Exclusive byte	STXRB
`STXRH`	Store Exclusive halfword	STXRH
`STXP`	Store Exclusive pair	STXP

C3.2.10 Load-Exclusive/Store-Exclusive unprivileged

The Load-Exclusive unprivileged and Store-Exclusive unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load-exclusive/store-exclusive unprivileged instructions support only one address mode:

- Base register with no offset.

See Load/store addressing modes.

Table C3-32 Load-exclusive/Store-exclusive unprivileged instructions

Mnemonic	Instruction	See
`LDTXR`	Load unprivileged exclusive register	LDTXR
`STTXR`	Store unprivileged exclusive register	STTXR

C3.2.11 Load-Acquire/Store-Release

The Load-Acquire, Load-AcquirePC and Store-Release instructions are added as part of the FEAT_LRCPC, FEAT_LRCPC2, and FEAT_LRCPC3 architecture features.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_LSE2 is not implemented, then:

- The Load-Acquire, Load-AcquirePC, and Store-Release instructions other than Load-Acquire pair and Store-Release pair require natural alignment, and an unaligned address generates an Alignment fault.
- Memory accesses generated by Load-Acquire pair or Store-Release pair instructions must be aligned to the size of the accessed element in the pair, otherwise the access generates an Alignment fault.

For more information on alignment requirements and behaviors, see Non-atomic Load-Acquire/Store-Release instructions.

The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of Load-Acquire, Load-AcquirePC, and Store-Release, see Load-Acquire, Load-AcquirePC, and Store-Release.

A Store-Release Exclusive instruction has the Release semantics only if the store is successful.

Armv8.1 adds more instructions with load-acquire and store-release mechanisms, see LoadLOAcquire/StoreLORelease.

Table C3-33 shows the Non-exclusive Load-Acquire/Store-Release instructions.

Table C3-33 Non-exclusive Load-Acquire and Store-Release instructions

Mnemonic	Instruction	See
LDAPR	Load-Acquire RCpc Register	LDAPR
LDAPRB	Load-Acquire RCpc Register Byte	LDAPRB
LDAPRH	Load-Acquire RCpc Register Halfword	LDAPRH
LDAPUR	Load-Acquire RCpc Register (unscaled)	LDAPUR
LDAPURB	Load-Acquire RCpc Register Byte (unscaled)	LDAPURB
LDAPURH	Load-Acquire RCpc Register Halfword (unscaled)	LDAPURH
LDAPURSB	Load-Acquire RCpc Register Signed Byte (unscaled) 32-bit	LDAPURSB
LDAPURSB	Load-Acquire RCpc Register Signed Byte (unscaled) 64-bit	LDAPURSB
LDAPURSH	Load-Acquire RCpc Register Signed Halfword (unscaled) 32-bit	LDAPURSH
LDAPURSH	Load-Acquire RCpc Register Signed Halfword (unscaled) 64-bit	LDAPURSH
LDAPURSW	Load-Acquire RCpc Register Signed Word (unscaled)	LDAPURSW
LDAR	Load-Acquire Register	LDAR
LDARB	Load-Acquire Byte	LDARB
LDARH	Load-Acquire Halfword	LDARH
STLR	Store-Release Register	STLR
STLRB	Store-Release Byte	STLRB
STLRH	Store-Release Halfword	STLRH
STLUR	Store-Release Register (unscaled)	STLUR
STLURB	Store-Release Register Byte (unscaled)	STLURB
STLURH	Store-Release Register Halfword (unscaled)	STLURH

Table C3-34 shows the Exclusive Load-Acquire/Store-Release instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-34 Exclusive Load-Acquire and Store-Release instructions

Mnemonic	Instruction	See
LDAXR	Load-Acquire Exclusive register	LDAXR
LDAXRB	Load-Acquire Exclusive byte	LDAXRB
LDAXRH	Load-Acquire Exclusive halfword	LDAXRH
LDAXP	Load-Acquire Exclusive pair	LDAXP
STLXR	Store-Release Exclusive register	STLXR
STLXRB	Store-Release Exclusive byte	STLXRB
STLXRH	Store-Release Exclusive halfword	STLXRH
STLXP	Store-Release Exclusive pair	STLXP

Table C3-35 Register Pair Ordered Load-AcquirePC and Store-Release instructions

Mnemonic	Instruction	See
LDIAPP	Load-Acquire RCpc Ordered Pair of Registers	LDIAPP
STILP	Store-Release Pair Ordered	STILP

C3.2.12 Load-Acquire/Store-Release unprivileged

The Load-Acquire unprivileged and Store-Release unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load-acquire/store-release unprivileged instructions support only one addressing mode:

- Base register with no offset.

See Load/store addressing modes.

Table C3-36 Load-Acquire/Store-Release unprivileged instructions

Mnemonic	Instruction	See
LDATXR	Load-acquire unprivileged exclusive register	LDATXR
STLTXR	Store-release unprivileged exclusive register	STLTXR

C3.2.13 LoadLOAcquire/StoreLORelease

The LoadLOAcquire/StoreLORelease instructions support only one addressing mode:

- Base register with no offset.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See Load/store addressing modes.

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of LoadLOAcquire/StoreLORelease, see LoadLOAcquire, StoreLORelease.

If FEAT_LSE2 is not implemented, then the LoadLOAcquire/StoreLORelease instructions require natural alignment, and an unaligned address generates an Alignment fault.

Table C3-37 shows the LoadLOAcquire/StoreLORelease instructions.

Table C3-37 LoadLOAcquire and StoreLORelease instructions

Mnemonic	Instruction	See
`LDLARB`	LoadLOAcquire byte	LDLARB
`LDLARH`	LoadLOAcquire halfword	LDLARH
`LDLAR`	LoadLOAcquire register	LDLAR
`STLLRB`	StoreLORelease byte	STLLRB
`STLLRH`	StoreLORelease halfword	STLLRH
`STLLR`	StoreLORelease register	STLLR

C3.2.14 Load/store scalar SIMD and floating-point

The load/store scalar SIMD and floating-point instructions operate on scalar values in the SIMD and floating-point register file as described in Advanced SIMD and floating-point scalar register names. The memory addressing modes available, described in Load/store addressing modes, are identical to the general-purpose register load/store instructions, and like those instructions permit arbitrary address alignment unless strict alignment checking is enabled. However, unlike the load/store instructions that transfer general-purpose registers, load/store scalar SIMD and floating-point instructions make no guarantee of atomicity, even when the address is naturally aligned to the size of the data.

C3.2.14.1 Load/store scalar SIMD and floating-point register

The load/store scalar SIMD and floating-point register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus unscaled 9-bit signed immediate offset.
- Base plus 64-bit register offset, optionally scaled.
- Base plus 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

For more information on the addressing modes, see Load/store addressing modes.

Note

The unscaled 9-bit signed immediate offset address mode requires its own instruction form, see Load/store scalar SIMD and floating-point register (unscaled offset).

Table C3-38 shows the load/store instructions for a single SIMD and floating-point register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-38 Load/store single SIMD and floating-point register instructions

Mnemonic	Instruction	See
LDR	Load scalar SIMD&FP register (register offset)	LDR (register, SIMD&FP)
	Load scalar SIMD&FP register (immediate offset)	LDR (immediate, SIMD&FP)
	Load scalar SIMD&FP register (PC-relative literal)	LDR (literal, SIMD&FP)
STR	Store scalar SIMD&FP register (register offset)	STR (register, SIMD&FP)
	Store scalar SIMD&FP register (immediate offset)	STR (immediate, SIMD&FP)

C3.2.14.2 Load/store scalar SIMD and floating-point register (unscaled offset)

The load /store scalar SIMD and floating-point register instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See also Load/store addressing modes.

The load/store scalar SIMD and floating-point register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store single SIMD and floating-point instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset. This is similar to the load/store register (unscaled offset) instructions, that disambiguate this instruction class from the load/store register instruction, see Load/store register (unscaled offset).

Table C3-39 shows the load/store SIMD and floating-point register instructions with an unscaled offset.

Table C3-39 Load/store SIMD and floating-point register instructions

Mnemonic	Instruction	See
LDUR	Load scalar SIMD&FP register (unscaled offset)	LDUR (SIMD&FP)
STUR	Store scalar SIMD&FP register (unscaled offset)	STUR (SIMD&FP)

C3.2.14.3 Load/store SIMD and floating-point register pair

The load/store SIMD and floating-point register pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

If a Load pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode and the register being loaded takes an UNKNOWN value.

For behavior when FEAT_LS64WB is implemented, see Requirements for single-copy atomicity.

Table C3-40 shows the load/store SIMD and floating-point register pair instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-40 Load/store SIMD and floating-point register pair instructions

Mnemonic	Instruction	See
LDP	Load pair of scalar SIMD&FP registers	LDP (SIMD&FP)
STP	Store pair of scalar SIMD&FP registers	STP (SIMD&FP)

C3.2.14.4 Load/store unprivileged SIMD and floating-point register pair

The load/store unprivileged SIMD and floating-point register pair instructions are added as part of the FEAT_LSUI architecture feature. These intructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load/store unprivileged SIMD and floating-point register pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

Table C3-41 Load/store unprivileged SIMD and floating-point register pair instructions

Mnemonic	Instruction	See
LDTP	Load unprivileged pair of scalar SIMD&FP registers	LDTP (SIMD&FP)
STTP	Store unprivileged pair of scalar SIMD&FP registers	STTP (SIMD&FP)

C3.2.14.5 Load/store SIMD and floating-point non-temporal pair

The load/store SIMD and floating-point non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be prefetched and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a load non-temporal pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a load non-temporal pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-42 shows the load/store SIMD and floating-point Non-temporal pair instructions.

Table C3-42 Load/store SIMD and floating-point non-temporal pair instructions

Mnemonic	Instruction	See
LDNP	Load pair of scalar SIMD&FP registers	LDNP (SIMD&FP)
STNP	Store pair of scalar SIMD&FP registers	STNP (SIMD&FP)

C3.2.14.6 Load/store unprivileged SIMD and floating-point non-temporal pair

The load/store unprivileged SIMD and floating-point non-temporal pair instructions are added as part of the FEAT_LSUI architecture feature. These intructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load/store SIMD and floating-point non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

Table C3-43 Load/store unprivileged SIMD and floating-point non-temporal pair instructions

Mnemonic	Instruction	See
LDTNP	Load Unprivileged Non-temporal Pair	LDTNP (SIMD&FP)
STTNP	Store Unprivileged Non-temporal Pair	STTNP (SIMD&FP)

C3.2.15 Load/store Advanced SIMD

The Advanced SIMD load/store structure instructions support the following addressing modes:

- Base register only.
- Post-indexed by a 64-bit register.
- Post-indexed by an immediate, equal to the number of bytes transferred.

Load/store vector instructions, like other load/store instructions, allow any address alignment, unless strict alignment checking is enabled. If strict alignment checking is enabled, then alignment checking to the size of the element is performed. However, unlike the load/store instructions that transfer general-purpose registers, the load/store vector instructions do not guarantee atomicity, even when the address is naturally aligned to the size of the element.

C3.2.15.1 Load/store structures

Table C3-44 shows the load/store structure instructions. A post-increment immediate offset, if present, must be 8, 16, 24, 32, 48, or 64, depending on the number of elements transferred.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-44 Load/store multiple structures instructions

Mnemonic	Instruction	See
LD1	Load single 1-element structure to one lane of one register	LD1 (single structure)
	Load multiple 1-element structures to one register or to two, three, or four consecutive registers	LD1 (multiple structures)
LD2	Load single 2-element structure to one lane of two consecutive registers	LD2 (single structure)
	Load multiple 2-element structures to two consecutive registers	LD2 (multiple structures)
LD3	Load single 3-element structure to one lane of three consecutive registers	LD3 (single structure)
	Load multiple 3-element structures to three consecutive registers	LD3 (multiple structures)
LD4	Load single 4-element structure to one lane of four consecutive registers	LD4 (single structure)
	Load multiple 4-element structures to four consecutive registers	LD4 (multiple structures)
ST1	Store single 1-element structure from one lane of one register	ST1 (single structure)
	Store multiple 1-element structures from one register, or from two, three, or four consecutive registers	ST1 (multiple structures)
ST2	Store single 2-element structure from one lane of two consecutive registers	ST2 (single structure)
	Store multiple 2-element structures from two consecutive registers	ST2 (multiple structures)
ST3	Store single 3-element structure from one lane of three consecutive registers	ST3 (single structure)
	Store multiple 3-element structures from three consecutive registers	ST3 (multiple structures)
ST4	Store single 4-element structure from one lane of four consecutive registers	ST4 (single structure)
	Store multiple 4-element structures from four consecutive registers	ST4 (multiple structures)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.15.2 Load single structure and replicate

Table C3-45 shows the Load single structure and replicate instructions. A post-increment immediate offset, if present, must be 1, 2, 3, 4, 6, 8, 12, 16, 24, or 32, depending on the number of elements transferred.

Table C3-45 Load single structure and replicate instructions

Mnemonic	Instruction	See
`LD1R`	Load single 1-element structure and replicate to all lanes of one register	LD1R
`LD2R`	Load single 2-element structure and replicate to all lanes of two registers	LD2R
`LD3R`	Load single 3-element structure and replicate to all lanes of three registers	LD3R
`LD4R`	Load single 4-element structure and replicate to all lanes of four registers	LD4R

Table C3-46 SIMD and floating-point Non-exclusive Load-AcquirePC and Store-Release instructions

Mnemonic	Instruction	See
`LDAP1 (SIMD&FP)`	Load-Acquire RCpc one single-element structure to one lane of one register	LDAP1 (SIMD&FP)
`LDAPUR (SIMD&FP)`	Load-Acquire RCpc SIMD&FP Register (unscaled offset)	LDAPUR (SIMD&FP)
`STL1 (SIMD&FP)`	Store-Release a single-element structure from one lane of one register	STL1 (SIMD&FP)
`STLUR (SIMD&FP)`	Store-Release SIMD&FP Register (unscaled offset)	STLUR (SIMD&FP)

C3.2.16 Prefetch memory

The Prefetch memory instructions support the following addressing modes:

- Base register with no offset.
- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset. This can be optionally scaled by 8-bits, for example `LSL`#3.
- Base plus a 32-bit extended register offset. This can be optionally scaled by 8-bits.
- PC-relative literal.

The Prefetch memory instructions signal to the memory system that memory accesses from a specified address or range of addresses are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory access when they do occur, such as prefetching the specified address or range of addresses into one or more caches. Because these signals are only hints, it is valid for the PE to treat any or all prefetch instructions as a `NOP`.

Because they are hints to the memory system, the operation of a prefetch memory instruction cannot cause a synchronous exception. However, a memory operation performed as a result of one of these memory system hints might in exceptional cases trigger an asynchronous event, and thereby influence the execution of the PE. An example of an asynchronous event that might be triggered is an SError exception.

A prefetch memory instruction can only have an effect on software visible structures as defined in the translation regime of the current Exception level, such as caches and translation lookaside buffers associated with memory locations that can be accessed by reads, writes, or execution.

A `PRFM`, `PRFUM`, or `RPRFM` instruction is guaranteed not to access Device memory.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A PRFM or PRFUM instruction using a PLI hint cannot result in any access that could not be performed by the PE speculatively fetching an instruction. Therefore, if all associated MMUs are disabled, a PLI hint can only access memory locations consistent with instruction fetches permitted by The effects of disabling an address translation stage.

Arm recommends that when a PRFM or a PRFUM instruction describes a particular cache level, then the memory location described at that cache level should be made available at that same cache level, even if the line had previously been at a level of the cache that is closer to the PE than the specified cache level.

Note

For a PRFM PST instruction, this might result in the demotion of a cache line from a level that was closer to the cache.

Table C3-47 shows the Prefetch memory instructions.

Table C3-47 Prefetch memory instructions

Mnemonic	Instruction	See
PRFM	Prefetch memory (register offset)	PRFM (register)
	Prefetch memory (immediate offset)	PRFM (immediate)
	Prefetch memory (PC-relative offset)	PRFM (literal)
PRFUM	Prefetch memory (unscaled offset)	PRFUM
RPRFM	Range prefetch memory	RPRFM

C3.2.17 Atomic instructions

The atomic instructions perform atomic read and write operations on a memory location such that the architecture guarantees that no modification of that memory location by another observer can occur between the read and the write defined by that instruction.

This section describes the following operations:

- Atomic integer memory operations.
- Atomic floating-point memory operations.
- Unprivileged atomic memory operations.
- Swap operations.
- Swap unprivileged.
- Compare and Swap operations.
- Compare and Swap unprivileged.
- Read-Check-Write.

The atomic instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

For the purpose of permission checking, and for watchpoints, all of the Atomic memory operation instructions are treated as performing both a load and a store.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the architecture. The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity. These operations map to the acquire and release definitions, and are counted as Load-Acquire and Store-Release operations respectively.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.17.1 Atomic integer memory operations

The address accessed by the 128-bit atomic instructions must be aligned to 128-bits, otherwise the instructions generate an Alignment fault.

For the `LD<OP>` instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

The `ST<OP>` instructions, and `LD<OP>` instructions where the destination register is WZR or XZR, are not regarded as doing a read for the purpose of a `DMB LD` barrier.

For information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-48 Atomic integer memory operation instructions

Mnemonic	Instruction	See
LDADD	Atomic add	LDADD, LDADDA, LDADDAL, LDADDL
LDADDB	Atomic add on byte	LDADDB, LDADDAB, LDADDALB, LDADDLB
LDADDH	Atomic add on halfword	LDADDH, LDADDAH, LDADDALH, LDADDLH
LDCLR	Atomic bit clear	LDCLR, LDCLRA, LDCLRAL, LDCLRL
LDCLRB	Atomic bit clear on byte	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB
LDCLRH	Atomic bit clear on halfword	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH
LDCLRP	Atomic bit clear on quadword	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL
LDEOR	Atomic exclusive-OR	LDEOR, LDEORA, LDEORAL, LDEORL
LDEORB	Atomic exclusive-OR on byte	LDEORB, LDEORAB, LDEORALB, LDEORLB
LDEORH	Atomic exclusive-OR on halfword	LDEORH, LDEORAH, LDEORALH, LDEORLH
LDSET	Atomic bit set	LDSET, LDSETA, LDSETAL, LDSETL
LDSETB	Atomic bit set on byte	LDSETB, LDSETAB, LDSETALB, LDSETLB
LDSETH	Atomic bit set on halfword	LDSETH, LDSETAH, LDSETALH, LDSETLH
LDSETP	Atomic bit set on quadword	LDSETP, LDSETPA, LDSETPAL, LDSETPL
LDMAX	Atomic signed maximum	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL
LDMAXB	Atomic signed maximum on byte	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB
LDMAXH	Atomic signed maximum on halfword	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH
LDMIN	Atomic signed minimum	LDSMIN, LDSMINA, LDSMINAL, LDSMINL
LDMINB	Atomic signed minimum on byte	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB
LDMINH	Atomic signed minimum on halfword	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH
LDUMAX	Atomic unsigned maximum	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL
LDUMAXB	Atomic unsigned maximum on byte	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB
LDUMAXH	Atomic unsigned maximum on halfword	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH
LDUMIN	Atomic unsigned minimum	LDUMIN, LDUMINA, LDUMINAL, LDUMINL
LDUMINB	Atomic unsigned minimum on byte	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB
LDUMINH	Atomic unsigned minimum on halfword	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
STADD	Atomic add, without return	STADD, STADDL
STADDB	Atomic add on byte, without return	STADDB, STADDLB
STADDH	Atomic add on halfword, without return	STADDH, STADDLH
STCLR	Atomic bit clear, without return	STCLR, STCLRL
STCLRB	Atomic bit clear on byte, without return	STCLRB, STCLRLB
STCLRH	Atomic bit clear on halfword, without return	STCLRH, STCLRLH
STEOR	Atomic exclusive-OR, without return	STEOR, STEORL
STEORB	Atomic exclusive-OR on byte, without return	STEORB, STEORLB
STEORH	Atomic exclusive-OR on halfword, without return	STEORH, STEORLH
STSET	Atomic bit set, without return	STSET, STSETL
STSETB	Atomic bit set on byte, without return	STSETB, STSETLB
STSETH	Atomic bit set on halfword, without return	STSETH, STSETLH
STMAX	Atomic signed maximum, without return	STSMAX, STSMAXL
STMAXB	Atomic signed maximum on byte, without return	STSMAXB, STSMAXLB
STMAXH	Atomic signed maximum on halfword, without return	STSMAXH, STSMAXLH
STMIN	Atomic signed minimum, without return	STSMIN, STSMINL
STMINB	Atomic signed minimum on byte, without return	STSMINB, STSMINLB
STMINH	Atomic signed minimum on halfword, without return	STSMINH, STSMINLH
STUMAX	Atomic unsigned maximum, without return	STUMAX, STUMAXL
STUMAXB	Atomic unsigned maximum on byte, without return	STUMAXB, STUMAXLB
STUMAXH	Atomic unsigned maximum on halfword, without return	STUMAXH, STUMAXLH
STUMIN	Atomic unsigned minimum, without return	STUMIN, STUMINL
STUMINB	Atomic unsigned minimum on byte, without return	STUMINB, STUMINLB
STUMINH	Atomic unsigned minimum on halfword, without return	STUMINH, STUMINLH

C3.2.17.2 Atomic floating-point memory operations

The atomic floating-point memory operation instructions atomically load a value from memory, perform a floating-point arithmetic operation, and store the result back to memory.

For more information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-49 Atomic floating-point memory instructions

Mnemonic	Instruction	See
LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL	Atomic BFloat16 add	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL
LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL	Atomic BFloat16 maximum	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL
LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML	Atomic BFloat16 maximum number	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL	Atomic BFloat16 minimum	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL
LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML	Atomic BFloat16 minimum number	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML
LDFADD, LDFADDA, LDFADDAL, LDFADDL	Atomic floating-point add	LDFADD, LDFADDA, LDFADDAL, LDFADDL
LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL	Atomic floating-point maximum	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL
LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML	Atomic floating-point maximum number	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML
LDFMIN, LDFMINA, LDFMINAL, LDFMINL	Atomic floating-point minimum	LDFMIN, LDFMINA, LDFMINAL, LDFMINL
LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML	Atomic floating-point minimum number	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML
STBFADD, STBFADDL	Atomic BFloat16 add, without return	STBFADD, STBFADDL
STBFMAX, STBFMAXL	Atomic BFloat16 maximum, without return	STBFMAX, STBFMAXL
STBFMAXNM, STBFMAXNML	Atomic BFloat16 maximum number, without return	STBFMAXNM, STBFMAXNML
STBFMIN, STBFMINL	Atomic BFloat16 minimum, without return	STBFMIN, STBFMINL
STBFMINNM, STBFMINNML	Atomic BFloat16 minimum number, without return	STBFMINNM, STBFMINNML
STFADD, STFADDL	Atomic floating-point add, without return	STFADD, STFADDL
STFMAX, STFMAXL	Atomic floating-point maximum, without return	STFMAX, STFMAXL
STFMAXNM, STFMAXNML	Atomic floating-point maximum number, without return	STFMAXNM, STFMAXNML
STFMIN, STFMINL	Atomic floating-point minimum, without return	STFMIN, STFMINL
STFMINNM, STFMINNML	Atomic floating-point minimum number, without return	STFMINNM, STFMINNML

C3.2.17.3 Unprivileged atomic memory operations

The unprivileged atomic memory operation instructions are added as part of the FEAT_LSUI architecture feature. These intructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The atomic memory operation instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-50 Unprivileged atomic memory operation instructions

Mnemonic	Instruction	See
LDTADD	Atomic add unprivileged	LDTADD, LDTADDA, LDTADDAL, LDTADDL
STTADD	Atomic add unprivileged, without return	STTADD, STTADDL
LDTSET	Atomic bit set unprivileged	LDTSET, LDTSETA, LDTSETAL, LDTSETL
STTSET	Atomic bit set unprivileged, without return	STTSET, STTSETL
LDTCLR	Atomic bit clear unprivileged	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL
STTCLR	Atomic bit clear unprivileged, without return	STTCLR, STTCLRL

C3.2.17.4 Swap operations

For the SWP instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

If the destination register Rt is WZR or XZR, this is not regarded as doing a read for the purpose of a DMB LD barrier.

For information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-51 Swap instructions

Mnemonic	Instruction	See
SWP	Swap	SWP, SWPA, SWPAL, SWPL
SWPB	Swap byte	SWPB, SWPAB, SWPALB, SWPLB
SWPH	Swap halfword	SWPH, SWPAH, SWPALH, SWPLH
SWPP	Swap quadword	SWPP, SWPPA, SWPPAL, SWPPL

C3.2.17.4.1 Swap unprivileged

The swap unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These intructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The swap unprivileged instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

Table C3-52 Swap unprivileged instructions

Mnemonic	Instruction	See
SWPT	Swap	SWPT, SWPTA, SWPTAL, SWPTL

C3.2.17.5 Compare and Swap operations

If FEAT_LSE2 is not implemented then:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CAS instructions require natural alignment.
- The CASP instructions require alignment to the total size of the memory being accessed.

If a compare and swap instruction does not perform a store, then the instruction does not have release semantics, regardless of the instruction ordering options.

For the CAS and CASP instructions, the architecture permits that a data read clears any Exclusives monitors associated with that location, even if the compare subsequently fails. If these instructions generate a synchronous Data Abort, the registers which are compared and loaded are restored to the values held in the registers before the instruction was executed.

If the destination register Rs is WZR or XZR, this is not regarded as doing a read for the purpose of a DMB LD barrier.

For more information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-53 Compare and swap instructions

Mnemonic	Instruction	See
CAS	Compare and swap	CAS, CASA, CASAL, CASL
CASB	Compare and swap byte	CASB, CASAB, CASALB, CASLB
CASH	Compare and swap halfword	CASH, CASAH, CASALH, CASLH
CASP	Compare and swap pair	CASP, CASPA, CASPAL, CASPL

C3.2.17.5.1 Compare and Swap unprivileged

The Compare and Swap unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These intructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The Compare and Swap unprivileged instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

Table C3-54 Compare and swap unprivileged instructions

Mnemonic	Instruction	See
CAST	Compare and swap unprivileged	CAST, CASAT, CASALT, CASLT
CASPT	Compare and swap pair unprivileged	CASPT, CASPAT, CASPALT, CASPLT

C3.2.17.6 Read-Check-Write

If FEAT_THE is implemented, then the Read-Check-Write instructions, RCW, and Read-Check-Write Software instructions, RCWS, are provided to conditionally update stage 1 descriptors.

RCW and RCWS instructions update PSTATE.{N, Z, C, V} to {0, 0, 1, 0}, with all of the following exceptions:

- For the compare and swap variants of the RCW and RCWS instructions, if the compare fails, then PSTATE.N is set to 1.
- Otherwise, all of the following apply:
  - — If RCW and RCWS instructions fail the RCW checks, then PSTATE.Z is set to 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If RCWS instructions fail the RCWS checks, then PSTATE.C is set to 0.

If FEAT_D128 is implemented, then RCW and RCWS instructions that operate on 128 bits of data are provided.

If the address is not aligned to the data transfer size, then RCW and RCWS instructions generate an Alignment fault.

The following table shows the RCW and RCWS instructions that operate on 64 bits of data:

Table C3-55 64 bit Read-Check-Write instructions

Mnemonic	Instruction	See
RCWCAS	Read Check Write Compare and Swap doubleword	RCWCAS, RCWCASA, RCWCASL, RCWCASAL
RCWSCAS	Read Check Write Software Compare and Swap doubleword	RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL
RCWCLR	Read Check Write Atomic bit clear on doubleword	RCWCLR, RCWCLRA, RCWCLRL, RCWCLRAL
RCWSCLR	Read Check Write Software Atomic bit clear on doubleword	RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL
RCWSET	Read Check Write Atomic bit set on doubleword	RCWSET, RCWSETA, RCWSETL, RCWSETAL
RCWSWP	Read Check Write Swap doubleword	RCWSWP, RCWSWPA, RCWSWPL, RCWSWPAL
RCWSSWP	Read Check Write Software Swap doubleword	RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL

The following table shows the RCW and RCWS instructions that operate on 128 bits of data:

Table C3-56 128 bit Read-Check-Write instructions

Mnemonic	Instruction	See
RCWCASP	Read Check Write Compare and Swap quadword	RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL
RCWSCASP	Read Check Write Software Compare and Swap quadword	RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL
RCWCLRP	Read Check Write Atomic bit clear on quadword	RCWCLRP, RCWCLRPA, RCWCLRPL, RCWCLRPAL
RCWSCLRP	Read Check Write Software Atomic bit clear on quadword	RCWSCLRP, RCWSCLRPA, RCWSCLRPL, RCWSCLRPAL
RCWSETP	Read Check Write Atomic bit set on quadword	RCWSETP, RCWSETPA, RCWSETPL, RCWSETPAL
RCWSWPP	Read Check Write Swap quadword	RCWSWPP, RCWSWPPA, RCWSWPPL, RCWSWPPAL
RCWSSWPP	Read Check Write Software Swap quadword	RCWSSWPP, RCWSSWPPA, RCWSSWPPL, RCWSSWPPAL

C3.2.18 Memory Tagging instructions

C3.2.18.1 Tag load

Table C3-57 shows the Memory Tagging Extension Tag load instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-57 Tag load instructions

Mnemonic	Instruction	See
LDG	Load Allocation Tag	LDG

C3.2.18.2 Tag store

Table C3-58 shows the Memory Tagging Extension Tag store instructions.

Also see DC GVA.

Table C3-58 Tag store instructions

Mnemonic	Instruction	See
STG	Store Allocation Tag	STG
ST2G	Store Allocation Tags	ST2G

C3.2.18.3 Tag and Data store

Table C3-59 shows the Memory Tagging Extension Tag and Data store instructions.

Also see DC GZVA and Table C3-63.

Table C3-59 Tag and Data store instructions

Mnemonic	Instruction	See
STZG	Store Allocation Tag, zeroing	STZG
STZ2G	Store Allocation Tags, zeroing	STZ2G
STGP	Store Allocation Tag and pair of registers	STGP

C3.2.18.4 Bulk Allocation tag access

Table C3-60 shows the Memory Tagging Extension Bulk Allocation tag access instructions.

Table C3-60 Bulk Allocation tag access instructions

Mnemonic	Instruction	See
LDGM	Load tag multiple	LDGM
STGM	Store Allocation Tag multiple	STGM
STZGM	Store Allocation Tag and zero multiple	STZGM

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.19 Memory Copy and Memory Set instructions

If FEAT_MOPS is implemented, the `CPY*`, `SETE*`, `SETM*`, and `SETP*` instructions described in this section are implemented. If FEAT_MOPS and FEAT_MTE are implemented, the `SETG*` instructions described in this section are also implemented. Collectively, the instructions described in this section are referred to as *Memory Copy and Memory Set* instructions.

To perform a memory copy or memory set, three instructions - a prologue, then a main, and then an epilogue - are expected to be run in succession, with no instruction appearing in the code between them:

- To perform a memory copy, forward-only: `CPYFP*`, then `CPYFM*`, and then `CPYFE*`.
- To perform a memory copy, forward or backward: `CPYP*`, then `CPYM*`, and then `CPYE*`.
- To perform a memory set with tag setting: `SETGP*`, then `SETGM*`, and then `SETGE*`.
- To perform a memory set without tag setting: `SETP*`, then `SETM*`, and then `SETE*`.

The variant of each instruction is also expected to be the same throughout one of these sequences. For example, `CPYFPWTWN`, `CPYFMWTWN`, `CPYFEWTWN`.

Note

The Memory Copy and Memory Set instructions are expected to be the preferred approach for compilation for performance for any of the following cases:

- The size or alignments of the copy or set operation are not known at compile time.
- The size or alignments of the copy or set operation are known at compile time, but not amenable to the operation being performed using load and store instructions without looping.

Arm recommends that hardware implementations optimise the performance of these cases.

Fetching of a Memory Copy and Memory Set instruction multiple times during its execution is permissible.

The handling of synchronous exceptions generated by Memory Copy and Memory Set instructions is described in ESR_ELx, FAR_ELx, Definition of a precise exception and imprecise exception, and Synchronous exception entry.

The handling of asynchronous exceptions generated by Memory Copy and Memory Set instructions is described in Definition of a precise exception and imprecise exception and Asynchronous exception entry.

Memory Copy and Memory Set exceptions can be generated by execution of Memory Copy and Memory Set instructions restarting on a physical hardware PE implementation that is different from the physical hardware PE implementation that an exception was taken from. For more information, see Memory Copy and Memory Set exceptions.

If an exception is taken during the execution of a Memory Copy and Memory Set instruction, that instruction has not been executed for the purposes of instruction counting, instruction tracing, statistical profiling, or single stepping.

For the purposes of single stepping and performance monitoring, each Memory Copy and Memory Set instruction is regarded as a single instruction that performs a store and, in the case of `CPY*` instructions, also performs a load.

For Memory Copy and Memory Set instructions, the following are not architecturally defined:

- The size of the memory transactions they create.
- The order between accesses to different addresses.

The `CPY*` instructions are guaranteed to make forward progress if none of the following leaf level translation table entries fault:

- The source leaf level translation table entry used to translate the address encoded in `Xs`.
- If the range of remaining source addresses, encoded in `Xs` and `Xn`, crosses a page boundary:
  - The next leaf level translation table entry, as determined by the copy direction, adjacent to the source leaf level translation table entry used to translate the address encoded in `Xs`.
- The destination leaf level translation table entry used to translate the address encoded in `Xd`.
- If the range of remaining destination addresses, encoded in `Xd` and `Xn`, crosses a page boundary:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The next leaf level translation table entry, as determined by the copy direction, adjacent to the destination leaf level translation table entry used to translate the address encoded in `Xd`.

The `SET*` instructions are guaranteed to make forward progress if none of the following leaf level translation table entries fault:

- The destination leaf level translation table entry used to translate the address encoded in `Xd`.
- If the the range of remaining addresses, encoded in `Xd` and `Xn`, crosses a page boundary:
  — The next leaf level translation table entry adjacent to the destination leaf level translation table entry used to translate the address encoded in `Xd`.

Note

For a `CPY*` instruction, destination address, source address and copy size are encoded in `Xd`, `Xs` and `Xn` depending on the Option and Direction chosen by the implementation.
For a `SET*` instruction, destination address and set size are encoded in `Xd` and `Xn` depending on the Option and Direction chosen by the implementation.
A PE can ensure that the forward progress described in this section can be achieved by ensuring that, when an MMU fault is encountered, all bytes leading up to the fault have been operated on and the operand registers have been updated.
A PE should not report MMU faults other than for the leaf level translation table entries described in this section.

Table C3-61 shows the memory copy, forward-only instructions.

Table C3-61 Memory copy, forward-only instructions

Mnemonic	Instruction	See
`CPYFE`	Memory Copy Forward-only Epilogue	CPYFP, CPYFM, CPYFE Epilogue variant
`CPYFEN`	Memory Copy Forward-only Epilogue, reads and writes non-temporal	CPYFPN, CPYFMN, CPYFEN Epilogue variant
`CPYFERN`	Memory Copy Forward-only Epilogue, reads non-temporal	CPYFPRN, CPYFMRN, CPYFERN Epilogue variant
`CPYFERT`	Memory Copy Forward-only Epilogue, reads unprivileged	CPYFPRT, CPYFMRT, CPYFERT Epilogue variant
`CPYFERTN`	Memory Copy Forward-only Epilogue, reads unprivileged, reads and writes non-temporal	CPYFPRTN, CPYFMRTN, CPYFERTN Epilogue variant
`CPYFERTRN`	Memory Copy Forward-only Epilogue, reads unprivileged and non-temporal	CPYFPRTRN, CPYFMRTRN, CPYFERTRN Epilogue variant
`CPYFERTWN`	Memory Copy Forward-only Epilogue, reads unprivileged, writes non-temporal	CPYFPRTWN, CPYFMRTWN, CPYFERTWN Epilogue variant
`CPYFET`	Memory Copy Forward-only Epilogue, reads and writes unprivileged	CPYFPT, CPYFMT, CPYFET Epilogue variant
`CPYFETN`	Memory Copy Forward-only Epilogue, reads and writes unprivileged and non-temporal	CPYFPTN, CPYFMTN, CPYFETN Epilogue variant
`CPYFETRN`	Memory Copy Forward-only Epilogue, reads and writes unprivileged, reads non-temporal	CPYFPTRN, CPYFMTRN, CPYFETRN Epilogue variant
`CPYFETWN`	Memory Copy Forward-only Epilogue, reads and writes unprivileged, writes non-temporal	CPYFPTWN, CPYFMTWN, CPYFETWN Epilogue variant
`CPYFEWN`	Memory Copy Forward-only Epilogue, writes non-temporal	CPYFPWN, CPYFMWN, CPYFEWN Epilogue variant
`CPYFEWT`	Memory Copy Forward-only Epilogue, writes unprivileged	CPYFPWT, CPYFMWT, CPYFEWT Epilogue variant

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CPYFEWTN	Memory Copy Forward-only Epilogue, writes unprivileged, reads and writes non-temporal	CPYFPWTN, CPYFMWTN, CPYFEWTN Epilogue variant
CPYFEWTRN	Memory Copy Forward-only Epilogue, writes unprivileged, reads non-temporal	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN Epilogue variant
CPYFMWTWN	Memory Copy Forward-only Epilogue, writes unprivileged and non-temporal	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN Epilogue variant
CPYFM	Memory Copy Forward-only Main	CPYFP, CPYFM, CPYFE Main variant
CPYFMN	Memory Copy Forward-only Main, reads and writes non-temporal	CPYFPN, CPYFMN, CPYFEN Main variant
CPYFMRN	Memory Copy Forward-only Main, reads non-temporal	CPYFPRN, CPYFMRN, CPYFERN Main variant
CPYFMRT	Memory Copy Forward-only Main, reads unprivileged	CPYFPRT, CPYFMRT, CPYFERT Main variant
CPYFMRTN	Memory Copy Forward-only Main, reads unprivileged, reads and writes non-temporal	CPYFPRTN, CPYFMRTN, CPYFERTN Main variant
CPYFMRTRN	Memory Copy Forward-only Main, reads unprivileged and non-temporal	CPYFPRTRN, CPYFMRTRN, CPYFERTRN Main variant
CPYFMRTWN	Memory Copy Forward-only Main, reads unprivileged, writes non-temporal	CPYFPRTWN, CPYFMRTWN, CPYFERTWN Main variant
CPYFMT	Memory Copy Forward-only Main, reads and writes unprivileged	CPYFPT, CPYFMT, CPYFET Main variant
CPYFMTN	Memory Copy Forward-only Main, reads and writes unprivileged and non-temporal	CPYFPTN, CPYFMTN, CPYFETN Main variant
CPYFMTRN	Memory Copy Forward-only Main, reads and writes unprivileged, reads non-temporal	CPYFPTRN, CPYFMTRN, CPYFETRN Main variant
CPYFMTWN	Memory Copy Forward-only Main, reads and writes unprivileged, writes non-temporal	CPYFPTWN, CPYFMTWN, CPYFETWN Main variant
CPYFMWN	Memory Copy Forward-only Main, writes non-temporal	CPYFPWN, CPYFMWN, CPYFEWN Main variant
CPYFMWT	Memory Copy Forward-only Main, writes unprivileged	CPYFPWT, CPYFMWT, CPYFEWT Main variant
CPYFMWTN	Memory Copy Forward-only Main, writes unprivileged, reads and writes non-temporal	CPYFPWTN, CPYFMWTN, CPYFEWTN Main variant
CPYFMWTRN	Memory Copy Forward-only Main, writes unprivileged, reads non-temporal	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN Main variant
CPYFMWTWN	Memory Copy Forward-only Main, writes unprivileged and non-temporal	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN Main variant
CPYFP	Memory Copy Forward-only Prologue	CPYFP, CPYFM, CPYFE Prologue variant
CPYFPN	Memory Copy Forward-only Prologue, reads and writes non-temporal	CPYFPN, CPYFMN, CPYFEN Prologue variant
CPYFPRN	Memory Copy Forward-only Prologue, reads non-temporal	CPYFPRN, CPYFMRN, CPYFERN Prologue variant
CPYFPRT	Memory Copy Forward-only Prologue, reads unprivileged	CPYFPRT, CPYFMRT, CPYFERT Prologue variant
CPYFPRTN	Memory Copy Forward-only Prologue, reads unprivileged, reads and writes non-temporal	CPYFPRTN, CPYFMRTN, CPYFERTN Prologue variant
CPYFPRTRN	Memory Copy Forward-only Prologue, reads unprivileged and non-temporal	CPYFPRTRN, CPYFMRTRN, CPYFERTRN Prologue variant

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CPYFPRTWN	Memory Copy Forward-only Prologue, reads unprivileged, writes non-temporal	CPYFPRTWN, CPYFMRTWN, CPYFERTWN Prologue variant
CPYFPT	Memory Copy Forward-only Prologue, reads and writes unprivileged	CPYFPT, CPYFMT, CPYFET Prologue variant
CPYFPTN	Memory Copy Forward-only Prologue, reads and writes unprivileged and non-temporal	CPYFPTN, CPYFMTN, CPYFETN Prologue variant
CPYFPTRN	Memory Copy Forward-only Prologue, reads and writes unprivileged, reads non-temporal	CPYFPTRN, CPYFMTRN, CPYFETRN Prologue variant
CPYFPTWN	Memory Copy Forward-only Prologue, reads and writes unprivileged, writes non-temporal	CPYFPTWN, CPYFMTWN, CPYFETWN Prologue variant
CPYFPWN	Memory Copy Forward-only Prologue, writes non-temporal	CPYFPWN, CPYFMWN, CPYFEWN Prologue variant
CPYFPWT	Memory Copy Forward-only Prologue, writes unprivileged	CPYFPWT, CPYFMWT, CPYFEWT Prologue variant
CPYFPWTN	Memory Copy Forward-only Prologue, writes unprivileged, reads and writes non-temporal	CPYFPWTN, CPYFMWTN, CPYFEWTN Prologue variant
CPYFPWTRN	Memory Copy Forward-only Prologue, writes unprivileged, reads non-temporal	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN Prologue variant
CPYFPWTWN	Memory Copy Forward-only Prologue, writes unprivileged and non-temporal	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN Prologue variant

Table C3-62 shows the memory copy, forward or backward instructions.

Table C3-62 Memory copy, forward or backward instructions

Mnemonic	Instruction	See
CPYE	Memory Copy Epilogue	CPYP, CPYM, CPYE Epilogue variant
CPYEN	Memory Copy Epilogue, reads and writes non-temporal	CPYPN, CPYMN, CPYEN Epilogue variant
CPYERN	Memory Copy Epilogue, reads non-temporal	CPYPRN, CPYMRN, CPYERN Epilogue variant
CPYERT	Memory Copy Epilogue, reads unprivileged	CPYPRT, CPYMRT, CPYERT Epilogue variant
CPYERTN	Memory Copy Epilogue, reads unprivileged, reads and writes non-temporal	CPYPRTN, CPYMRTN, CPYERTN Epilogue variant
CPYERTRN	Memory Copy Epilogue, reads unprivileged and non-temporal	CPYPRTRN, CPYMRTRN, CPYERTRN Epilogue variant
CPYERTWN	Memory Copy Epilogue, reads unprivileged, writes non-temporal	CPYPRTWN, CPYMRTWN, CPYERTWN Epilogue variant
CPYET	Memory Copy Epilogue, reads and writes unprivileged	CPYPT, CPYMT, CPYET Epilogue variant
CPYETN	Memory Copy Epilogue, reads and writes unprivileged and non-temporal	CPYPTN, CPYMTN, CPYETN Epilogue variant
CPYETRN	Memory Copy Epilogue, reads and writes unprivileged, reads non-temporal	CPYPTRN, CPYMTRN, CPYETRN Epilogue variant
CPYETWN	Memory Copy Epilogue, reads and writes unprivileged, writes non-temporal	CPYPTWN, CPYMTWN, CPYETWN Epilogue variant

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CPYEWN	Memory Copy Epilogue, writes non-temporal	CPYPWN, CPYMWN, CPYEWN Epilogue variant
CPYEWT	Memory Copy Epilogue, writes unprivileged	CPYPWT, CPYMWT, CPYEWT Epilogue variant
CPYEWTN	Memory Copy Epilogue, writes unprivileged, reads and writes non-temporal	CPYPWTN, CPYMWTN, CPYEWTN Epilogue variant
CPYEWTRN	Memory Copy Epilogue, writes unprivileged, reads non-temporal	CPYPWTRN, CPYMWTRN, CPYEWTRN Epilogue variant
CPYMWTWN	Memory Copy Epilogue, writes unprivileged and non-temporal	CPYPWTWN, CPYMWTWN, CPYEWTWN Epilogue variant
CPYM	Memory Copy Main	CPYP, CPYM, CPYE Main variant
CPYMN	Memory Copy Main, reads and writes non-temporal	CPYPN, CPYMN, CPYEN Main variant
CPYMRN	Memory Copy Main, reads non-temporal	CPYPRN, CPYMRN, CPYERN Main variant
CPYMRT	Memory Copy Main, reads unprivileged	CPYPRT, CPYMRT, CPYERT Main variant
CPYMRTN	Memory Copy Main, reads unprivileged, reads and writes non-temporal	CPYPRTN, CPYMRTN, CPYERTN Main variant
CPYMRTRN	Memory Copy Main, reads unprivileged and non-temporal	CPYPRTRN, CPYMRTRN, CPYERTRN Main variant
CPYMRTWN	Memory Copy Main, reads unprivileged, writes non-temporal	CPYPRTWN, CPYMRTWN, CPYERTWN Main variant
CPYMT	Memory Copy Main, reads and writes unprivileged	CPYPT, CPYMT, CPYET Main variant
CPYMTN	Memory Copy Main, reads and writes unprivileged and non-temporal	CPYPTN, CPYMTN, CPYETN Main variant
CPYMTRN	Memory Copy Main, reads and writes unprivileged, reads non-temporal	CPYPTRN, CPYMTRN, CPYETRN Main variant
CPYMTWN	Memory Copy Main, reads and writes unprivileged, writes non-temporal	CPYPTWN, CPYMTWN, CPYETWN Main variant
CPYMWN	Memory Copy Main, writes non-temporal	CPYPWN, CPYMWN, CPYEWN Main variant
CPYMWT	Memory Copy Main, writes unprivileged	CPYPWT, CPYMWT, CPYEWT Main variant
CPYMWTN	Memory Copy Main, writes unprivileged, reads and writes non-temporal	CPYPWTN, CPYMWTN, CPYEWTN Main variant
CPYMWTRN	Memory Copy Main, writes unprivileged, reads non-temporal	CPYPWTRN, CPYMWTRN, CPYEWTRN Main variant
CPYMWTWN	Memory Copy Main, writes unprivileged and non-temporal	CPYPWTWN, CPYMWTWN, CPYEWTWN Main variant
CPYP	Memory Copy Prologue	CPYP, CPYM, CPYE Prologue variant
CPYPN	Memory Copy Prologue, reads and writes non-temporal	CPYPN, CPYMN, CPYEN Prologue variant
CPYPRN	Memory Copy Prologue, reads non-temporal	CPYPRN, CPYMRN, CPYERN Prologue variant
CPYPRT	Memory Copy Prologue, reads unprivileged	CPYPRT, CPYMRT, CPYERT Prologue variant
CPYPRTN	Memory Copy Prologue, reads unprivileged, reads and writes non-temporal	CPYPRTN, CPYMRTN, CPYERTN Prologue variant
CPYPRTRN	Memory Copy Prologue, reads unprivileged and non-temporal	CPYPRTRN, CPYMRTRN, CPYERTRN Prologue variant
CPYPRTWN	Memory Copy Prologue, reads unprivileged, writes non-temporal	CPYPRTWN, CPYMRTWN, CPYERTWN Prologue variant
CPYPT	Memory Copy Prologue, reads and writes unprivileged	CPYPT, CPYMT, CPYET Prologue variant

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CPYPTN	Memory Copy Prologue, reads and writes unprivileged and non-temporal	CPYPTN, CPYMTN, CPYETN Prologue variant
CPYPTRN	Memory Copy Prologue, reads and writes unprivileged, reads non-temporal	CPYPTRN, CPYMTRN, CPYETRN Prologue variant
CPYPTWN	Memory Copy Prologue, reads and writes unprivileged, writes non-temporal	CPYPTWN, CPYMTWN, CPYETWN Prologue variant
CPYPWN	Memory Copy Prologue, writes non-temporal	CPYPWN, CPYMWN, CPYEWN Prologue variant
CPYPWT	Memory Copy Prologue, writes unprivileged	CPYPWT, CPYMWT, CPYEWT Prologue variant
CPYPWTN	Memory Copy Prologue, writes unprivileged, reads and writes non-temporal	CPYPWTN, CPYMWTN, CPYEWTN Prologue variant
CPYPWTRN	Memory Copy Prologue, writes unprivileged, reads non-temporal	CPYPWTRN, CPYMWTRN, CPYEWTRN Prologue variant
CPYPWTWN	Memory Copy Prologue, writes unprivileged and non-temporal	CPYPWTWN, CPYMWTWN, CPYEWTWN Prologue variant

Table C3-63 shows the memory set with tag setting instructions.

Table C3-63 Memory set with tag setting instructions

Mnemonic	Instruction	See
SETGE	Memory Set with tag setting Epilogue	SETGP, SETGM, SETGE Epilogue variant
SETGEN	Memory Set with tag setting Epilogue, non-temporal	SETGPN, SETGMN, SETGEN Epilogue variant
SETGET	Memory Set with tag setting Epilogue, unprivileged	SETGPT, SETGMT, SETGET Epilogue variant
SETGETN	Memory Set with tag setting Epilogue, unprivileged and non-temporal	SETGPTN, SETGMTN, SETGETN Epilogue variant
SETGM	Memory Set with tag setting Main	SETGP, SETGM, SETGE Main variant
SETGMN	Memory Set with tag setting Main, non-temporal	SETGPN, SETGMN, SETGEN Main variant
SETGMT	Memory Set with tag setting Main, unprivileged	SETGPT, SETGMT, SETGET Main variant
SETGMTN	Memory Set with tag setting Main, unprivileged and non-temporal	SETGPTN, SETGMTN, SETGETN Main variant
SETGP	Memory Set with tag setting Prologue	SETGP, SETGM, SETGE Prologue variant
SETGPN	Memory Set with tag setting Prologue, non-temporal	SETGPN, SETGMN, SETGEN Prologue variant
SETGPT	Memory Set with tag setting Prologue, unprivileged	SETGPT, SETGMT, SETGET Prologue variant
SETGPTN	Memory Set with tag setting Prologue, unprivileged and non-temporal	SETGPTN, SETGMTN, SETGETN Prologue variant

Table C3-64 shows the memory set without tag setting instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-64 Memory set without tag setting instructions

Mnemonic	Instruction	See
`SETE`	Memory Set Epilogue	SETP, SETM, SETE Epilogue variant
`SETEN`	Memory Set Epilogue, non-temporal	SETPN, SETMN, SETEN Epilogue variant
`SETET`	Memory Set Epilogue, unprivileged	SETPT, SETMT, SETET Epilogue variant
`SETETN`	Memory Set Epilogue, unprivileged and non-temporal	SETPTN, SETMTN, SETETN Epilogue variant
`SETM`	Memory Set Main	SETP, SETM, SETE Main variant
`SETMN`	Memory Set Main, non-temporal	SETPN, SETMN, SETEN Main variant
`SETMT`	Memory Set Main, unprivileged	SETPT, SETMT, SETET Main variant
`SETMTN`	Memory Set Main, unprivileged and non-temporal	SETPTN, SETMTN, SETETN Main variant
`SETP`	Memory Set Prologue	SETP, SETM, SETE Prologue variant
`SETPN`	Memory Set Prologue, non-temporal	SETPN, SETMN, SETEN Prologue variant
`SETPT`	Memory Set Prologue, unprivileged	SETPT, SETMT, SETET Prologue variant
`SETPTN`	Memory Set Prologue, unprivileged and non-temporal	SETPTN, SETMTN, SETETN Prologue variant

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.3 Loads and stores - SVE, SVE2

SVE vector load and store instructions transfer data in memory to or from elements of one or more vector or predicate transfer registers. SVE also includes vector prefetch instructions that provide read and write hints to the memory system. For SVE predicated load, store, and prefetch instructions, the memory element access size and type that is associated with each vector element is specified by a suffix to the instruction mnemonic, independently of the element size of the transfer registers. For example, LD1SH. The following table shows the supported instruction suffixes for SVE load, store, and prefetch instructions:

Instruction suffix	Memory element access size and type
B	Unsigned byte
H	Unsigned halfword or half-precision floating-point
W	Unsigned word or single-precision floating-point
D	Unsigned doubleword or double-precision floating-point
SB	Signed byte
SH	Signed halfword
SW	Signed word

The element size of the transfer registers is always greater than or equal to the memory element access size. When the element size of the transfer registers is strictly greater than the memory element access size, then these are referred to as unpacked data accesses. In the case of unpacked data accesses:

- For load instructions, each element access is sign-extended or zero-extended to fill the vector element, according to its memory element access size and type.
- For store instructions, each vector element is truncated to the memory element access size.

Where the vector element size and the memory element access size are the same, then these are referred to as packed data accesses. Signed access types are not supported for packed data accesses. Packed and unpacked access sizes and types relate to the vector element size of the transfer registers, as defined in the following table:

Vector element	Packed access suffix	Unpacked access suffixes
.B	B	-
.H	H	B, SB
.S	W	H, SH, B, SB
.D	D	W, SW, H, SH, B, SB

For gather-load and scatter-store instructions, the vector element size can only be .S or .D. This means that any non-contiguous memory element access of less than a word is unpacked. Non-contiguous memory element accesses of a word can be either packed or unpacked, depending on the vector element size.

Load, store, and prefetch instructions consist of the following:

- Predicated single vector contiguous element accesses.
- Predicated multiple vector contiguous structure load/store.
- Predicated non-contiguous element accesses.
- Predicated replicating element loads.
- Unpredicated vector register load/store.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Unpredicated predicate register load/store.
- Non-temporal gather/scatter.

All predicated load instructions zero the Inactive elements of the destination vector, except for Non-fault loads and First-fault loads when the corresponding FFR element is FALSE. When this happens, the content of the corresponding destination vector element is CONSTRAINED UNPREDICTABLE, as described in rule R_{NGFTJ}.

Prefetch instructions provide hints to hardware and do not change architectural state. Therefore, a *Governing predicate* for a prefetch instruction provides an additional hint which indicates the memory locations to be prefetched. Prefetch instructions require a prefetch operation specifier. SVE prefetch instructions support all of the prefetch operations except for the PLI prefetch operand types.

Load, store, and prefetch instructions that multiply a scalar index register or an index vector element by the memory element access size specify a shift type, followed by a shift amount in bits. The shift type can be one of `LSL`, `SXTW`, or `UXTW`. The shift amount is always Log_2 of the memory element access size, in bytes. The shift amount defaults to zero when the memory element access size is a byte, and the shift size can be omitted. The shift type of `LSL` must be omitted if the shift amount is omitted.

When included as part of the assembler syntax for an instruction, `MUL VL` indicates that the specified immediate index value is multiplied by the size of the addressed vector or predicate in memory, measured in bytes, irrespective of predication. For a detailed description of the meaning of this assembler syntax for each instruction, see the appropriate subsection below. When used in pseudocode, the symbol `VL` represents the vector length, measured in bits.

SVE load, store, and prefetch instructions do not support pre-indexed or post-indexed addressing.

C3.3.1 Predicated single vector contiguous element accesses

Predicated contiguous load and store instructions access memory locations starting from an address that is defined by a scalar base register plus either:

- A scalar index register.
- An immediate index value that is in the range -8 to 7, inclusive. This defaults to zero if omitted.

The predicated contiguous load and store instructions have two supporting addressing modes:

- Scalar base plus immediate index.
- Scalar base plus scalar index.

Predicated contiguous prefetch instructions address memory locations in a similar manner, with the index being either:

- A scalar index register.
- An immediate index value that is in the range of -32 to 31, inclusive. This defaults to zero if omitted.

For predicated contiguous load and store SVE instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of vector elements, irrespective of predication, and then multiplied by the memory element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.
- The scalar index register value is multiplied by the memory element access size in bytes. The index value is incremented by one after each element access, but the scalar index register is not updated by the instruction.
- When alignment checking is enabled for loads and stores, the value of the base address register must be aligned to the memory element access size.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-67 Predicated single vector contiguous element accesses

Mnemonic	Instruction	See
LD1	Contiguous load unsigned bytes to vector (scalar plus immediate, scalar plus scalar)	LD1B (scalar plus immediate, single register) LD1B (scalar plus scalar, single register)
	Contiguous load doublewords to vector (scalar plus immediate, scalar plus scalar)	LD1D (scalar plus immediate, single register) LD1D (scalar plus scalar, single register)
	Contiguous load unsigned halfwords to vector (scalar plus immediate, scalar plus scalar)	LD1H (scalar plus immediate, single register) LD1H (scalar plus scalar, single register)
	Contiguous load signed bytes to vector (scalar plus immediate)	LD1SB (scalar plus immediate)
	Contiguous load signed halfwords to vector (scalar plus immediate)	LD1SH (scalar plus immediate)
	Contiguous load signed words to vector (scalar plus immediate)	LD1SW (scalar plus immediate)
	Contiguous load unsigned words to vector (scalar plus immediate)	LD1W (scalar plus immediate, single register)
ST1	Contiguous store bytes from vector (scalar plus immediate, scalar plus scalar)	ST1B (scalar plus immediate, single register) ST1B (scalar plus scalar, single register)
	Contiguous store doublewords from vector (scalar plus immediate, scalar plus scalar)	ST1D (scalar plus immediate, single register) ST1D (scalar plus scalar, single register)
	Contiguous store halfwords from vector (scalar plus immediate, scalar plus scalar)	ST1H (scalar plus immediate, single register) ST1H (scalar plus scalar, single register)
	Contiguous store words from vector (scalar plus immediate, scalar plus scalar)	ST1W (scalar plus immediate, single register) ST1W (scalar plus scalar, single register)
LDFF1	Contiguous load first-fault unsigned bytes to vector (scalar plus scalar)	LDFF1B (scalar plus scalar)
	Contiguous load first-fault doublewords to vector (scalar plus scalar)	LDFF1D (scalar plus scalar)
	Contiguous load first-fault unsigned halfwords to vector (scalar plus scalar)	LDFF1H (scalar plus scalar)
	Contiguous load first-fault signed bytes to vector (scalar plus scalar)	LDFF1SB (scalar plus scalar)
	Contiguous load first-fault signed halfwords to vector (scalar plus scalar)	LDFF1SH (scalar plus scalar)
	Contiguous load first-fault signed words to vector (scalar plus scalar)	LDFF1SW (scalar plus scalar)
	Contiguous load first-fault unsigned words to vector (scalar plus scalar)	LDFF1W (scalar plus scalar)
LDNF1	Contiguous load non-fault unsigned bytes to vector	LDNF1B
	Contiguous load non-fault doublewords to vector	LDNF1D
	Contiguous load non-fault unsigned halfwords to vector	LDNF1H
	Contiguous load non-fault signed bytes to vector	LDNF1SB
	Contiguous load non-fault signed halfwords to vector	LDNF1SH
	Contiguous load non-fault signed words to vector	LDNF1SW
	Contiguous load non-fault unsigned words to vector	LDNF1W
LDNT1	Contiguous load non-temporal bytes to vector (scalar plus immediate, scalar plus scalar)	LDNT1B (scalar plus immediate, single register) LDNT1B (scalar plus scalar, single register)
	Contiguous load non-temporal doublewords to vector (scalar plus immediate, scalar plus scalar)	LDNT1D (scalar plus immediate, single register) LDNT1D (scalar plus scalar, single register)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Contiguous load non-temporal halfwords to vector (scalar plus immediate, scalar plus scalar)	LDNT1H (scalar plus immediate, single register) LDNT1H (scalar plus scalar, single register)
	Contiguous load non-temporal words to vector (scalar plus immediate, scalar plus scalar)	LDNT1W (scalar plus immediate, single register) LDNT1W (scalar plus scalar, single register)
STNT1	Contiguous store non-temporal bytes from vector (scalar plus immediate, scalar plus scalar)	STNT1B (scalar plus immediate, single register) STNT1B (scalar plus scalar, single register)
	Contiguous store non-temporal doublewords from vector (scalar plus immediate, scalar plus scalar)	STNT1D (scalar plus immediate, single register) STNT1D (scalar plus scalar, single register)
	Contiguous store non-temporal halfwords from vector (scalar plus immediate, scalar plus scalar)	STNT1H (scalar plus immediate, single register) STNT1H (scalar plus scalar, single register)
	Contiguous store non-temporal words from vector (scalar plus immediate, scalar plus scalar)	STNT1W (scalar plus immediate, single register) STNT1W (scalar plus scalar, single register)
PRF	Contiguous prefetch bytes (scalar plus immediate, scalar plus scalar)	PRFB (scalar plus immediate) PRFB (scalar plus scalar)
	Contiguous prefetch doublewords (scalar plus immediate, scalar plus scalar)	PRFD (scalar plus immediate) PRFD (scalar plus scalar)
	Contiguous prefetch halfwords (scalar plus immediate, scalar plus scalar)	PRFH (scalar plus immediate) PRFH (scalar plus scalar)
	Contiguous prefetch words (scalar plus immediate, scalar plus scalar)	PRFW (scalar plus immediate) PRFW (scalar plus scalar)

C3.3.2 Predicated multiple vector contiguous structure load/store

Predicated multiple vector contiguous structure load/store instructions are defined by a scalar base register plus either:

- A scalar index register.
- An immediate index that is a multiple of N, in the range -8xN to 7xN, inclusive. This defaults to zero if omitted.

The predicated contiguous structure load and store instructions have two supporting addressing modes:

- Scalar base plus immediate index.
- Scalar base plus scalar index.

For the predicated multiple vector contiguous structure load/store SVE instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of vector elements, irrespective of predication, and then multiplied by the memory element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.
- The scalar index register value is multiplied by the memory element access size in bytes. Following each element access, the index value is incremented by one but the instruction does not update the scalar index register.
- Each predicate element applies to a single structure in memory, or equivalently to the same element number within each of the two, three, or four transferred vector registers.
- These instructions support packed data accesses only.
- When alignment checking is enabled for loads and stores, the base address must be aligned to the element access size.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-68 Predicated multiple vector contiguous structure load/store

Mnemonic	Instruction	See
LD2	Contiguous load 2-byte structures to two vectors (scalar plus immediate, scalar plus scalar)	LD2B (scalar plus immediate) LD2B (scalar plus scalar)
	Contiguous load two-doubleword structures to two vectors (scalar plus immediate, scalar plus scalar)	LD2D (scalar plus immediate) LD2D (scalar plus scalar)
	Contiguous load two-halfword structures to two vectors (scalar plus immediate, scalar plus scalar)	LD2H (scalar plus immediate) LD2H (scalar plus scalar)
	Contiguous load two-word structures to two vectors (scalar plus immediate, scalar plus scalar)	LD2W (scalar plus immediate) LD2W (scalar plus scalar)
LD3	Contiguous load 3-byte structures to three vectors (scalar plus immediate, scalar plus scalar)	LD3B (scalar plus immediate) LD3B (scalar plus scalar)
	Contiguous load three-doubleword structures to three vectors (scalar plus immediate, scalar plus scalar)	LD3D (scalar plus immediate) LD3D (scalar plus scalar)
	Contiguous load three-halfword structures to three vectors (scalar plus immediate, scalar plus scalar)	LD3H (scalar plus immediate) LD3H (scalar plus scalar)
	Contiguous load three-word structures to three vectors (scalar plus immediate, scalar plus scalar)	LD3W (scalar plus immediate) LD3W (scalar plus scalar)
LD4	Contiguous load 4-byte structures to four vectors (scalar plus immediate, scalar plus scalar)	LD4B (scalar plus immediate) LD4B (scalar plus scalar)
	Contiguous load four-doubleword structures to four vectors (scalar plus immediate, scalar plus scalar)	LD4D (scalar plus immediate) LD4D (scalar plus scalar)
	Contiguous load four-halfword structures to four vectors (scalar plus immediate, scalar plus scalar)	LD4H (scalar plus immediate) LD4H (scalar plus scalar)
	Contiguous load four-word structures to four vectors (scalar plus immediate, scalar plus scalar)	LD4W (scalar plus immediate) LD4W (scalar plus scalar)
ST2	Contiguous store 2-byte structures from two vectors (scalar plus immediate, scalar plus scalar)	ST2B (scalar plus immediate) ST2B (scalar plus scalar)
	Contiguous store two-doubleword structures from two vectors (scalar plus immediate, scalar plus scalar)	ST2D (scalar plus immediate) ST2D (scalar plus scalar)
	Contiguous store two-halfword structures from two vectors (scalar plus immediate, scalar plus scalar)	ST2H (scalar plus immediate) ST2H (scalar plus scalar)
	Contiguous store two-word structures from two vectors (scalar plus immediate, scalar plus scalar)	ST2W (scalar plus immediate) ST2W (scalar plus scalar)
ST3	Contiguous store 3-byte structures from three vectors (scalar plus immediate, scalar plus scalar)	ST3B (scalar plus immediate) ST3B (scalar plus scalar)
	Contiguous store three-doubleword structures from three vectors (scalar plus immediate, scalar plus scalar)	ST3D (scalar plus immediate) ST3D (scalar plus scalar)
	Contiguous store three-halfword structures from three vectors (scalar plus immediate, scalar plus scalar)	ST3H (scalar plus immediate) ST3H (scalar plus scalar)
	Contiguous store three-word structures from three vectors (scalar plus immediate, scalar plus scalar)	ST3W (scalar plus immediate) ST3W (scalar plus scalar)
ST4	Contiguous store 4-byte structures from four vectors (scalar plus immediate, scalar plus scalar)	ST4B (scalar plus immediate) ST4B (scalar plus scalar)
	Contiguous store four-doubleword structures from four vectors (scalar plus immediate, scalar plus scalar)	ST4D (scalar plus immediate) ST4D (scalar plus scalar)
	Contiguous store four-halfword structures from four vectors (scalar plus immediate, scalar plus scalar)	ST4H (scalar plus immediate) ST4H (scalar plus scalar)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Contiguous store four-word structures from four vectors (scalar plus immediate, scalar plus scalar)	ST4W (scalar plus immediate) ST4W (scalar plus scalar)

C3.3.3 Predicated non-contiguous element accesses

Predicated non-contiguous element accesses address non-contiguous memory locations that are specified by either:

- A scalar base register plus a vector of indices or offsets.
- A vector of base addresses plus an immediate byte offset. The immediate byte offset is a multiple of the memory element access size, in the range 0 to 31 times the memory element access size, inclusive, and defaults to zero if omitted.

The predicated non-contiguous element accesses have two supporting addressing modes:

- Scalar base plus 64-bit vector index.
- Scalar base plus 32-bit vector index.
- Vector base plus immediate offset.

For this group of SVE instructions:

- Vector registers used as part of the address must specify a vector element size of 32 bits or 64 bits, .S or .D. For load and store instructions, the transfer register must specify the same vector element size.
- If the index vector register contains 32-bit index values then the lowest 32 bits of each index vector element can either be zero-extended or sign-extended to 64 bits.
- For load and store instructions, the index vector elements are then optionally multiplied by the memory element access size, in bytes, if a shift amount is specified. For prefetch instructions, the index vector elements are always multiplied by the memory element access size, in bytes.
- When alignment checking is enabled for loads and stores, the computed virtual address of each element must be aligned to the memory element access size.

Table C3-69 Predicated non-contiguous element accesses

Mnemonic	Instructions	See
`LD1`	Gather load unsigned bytes to vector (scalar plus vector, vector plus immediate)	LD1B (scalar plus vector) LD1B (vector plus immediate)
	Gather load doublewords to vector (scalar plus vector, vector plus immediate)	LD1D (scalar plus vector) LD1D (vector plus immediate)
	Gather load unsigned halfwords (scalar plus vector, vector plus immediate)	LD1H (scalar plus vector) LD1H (vector plus immediate)
	Gather load signed bytes to vector (scalar plus vector, vector plus immediate)	LD1SB (scalar plus vector) LD1SB (vector plus immediate)
	Gather load signed halfwords to vector (scalar plus vector, vector plus immediate)	LD1SH (scalar plus vector) LD1SH (vector plus immediate)
	Gather load unsigned words to vector (scalar plus vector, vector plus immediate)	LD1SW (scalar plus vector) LD1SW (vector plus immediate)
`ST1`	Scatter store bytes from a vector (scalar plus vector, vector plus immediate)	ST1B (scalar plus vector) ST1B (vector plus immediate)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instructions	See
	Scatter store doublewords from a vector (scalar plus vector, vector plus immediate)	ST1D (scalar plus vector) ST1D (vector plus immediate)
	Scatter store halfwords from a vector (scalar plus vector, vector plus immediate)	ST1H (scalar plus vector) ST1H (vector plus immediate)
	Scatter store words from a vector (scalar plus vector, vector plus immediate)	ST1W (scalar plus vector) ST1W (vector plus immediate)
`LDFF1`	Gather load first-fault unsigned bytes to vector (scalar plus vector, vector plus immediate)	LDFF1B (scalar plus vector) LDFF1B (vector plus immediate)
	Gather load first-fault doublewords to vector (scalar plus vector, vector plus immediate)	LDFF1D (scalar plus vector) LDFF1D (vector plus immediate)
	Gather load first-fault unsigned halfwords to vector (scalar plus vector, vector plus immediate)	LDFF1H (scalar plus vector) LDFF1SH (vector plus immediate)
	Gather load first-fault signed bytes to vector (scalar plus vector, vector plus immediate)	LDFF1SB (scalar plus vector) LDFF1SB (vector plus immediate)
	Gather load first-fault signed halfwords to vector (scalar plus vector, vector plus immediate)	LDFF1SH (scalar plus vector) LDFF1SH (vector plus immediate)
	Gather load first-fault signed words to vector (scalar plus vector, vector plus immediate)	LDFF1SW (scalar plus vector) LDFF1SW (vector plus immediate)
`LDFF1`	Gather load first-fault unsigned words to vector (scalar plus vector, vector plus immediate)	LDFF1W (scalar plus vector) LDFF1W (vector plus immediate)
`PRF`	Gather prefetch bytes (scalar plus vector, vector plus immediate)	PRFB (scalar plus vector) PRFB (vector plus immediate)
	Gather prefetch doublewords (scalar plus vector, vector plus immediate)	PRFD (scalar plus vector) PRFD (vector plus immediate)
	Gather prefetch halfwords (scalar plus vector, vector plus immediate)	PRFH (scalar plus vector) PRFH (vector plus immediate)
	Gather prefetch words (scalar plus vector, vector plus immediate)	PRFW (scalar plus vector) PRFW (vector plus immediate)

C3.3.4 Predicated replicating element loads

The load and replicate instructions read one or more contiguous memory locations starting from an address that is defined by a scalar base register plus either:

- A scalar index register.
- An immediate byte offset.

The predicated contiguous load and store instructions have two supporting addressing modes:

- Scalar base plus immediate offset.
- Scalar base plus scalar index.

This defaults to zero if omitted. For predicated replicating element load SVE instructions:

- When alignment checking is enabled, the base address must be aligned to the memory element access size.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-70 Predicated replicating element loads

Mnemonic	Instructions	See
LD1RB	Load and broadcast unsigned byte to vector	LD1RB
LD1RD	Load and broadcast doubleword to vector	LD1RD
LD1RH	Load and broadcast unsigned halfword to vector	LD1RH
LD1ROB	Contiguous load and replicate 32-bytes (scalar plus immediate, scalar plus scalar)	LD1ROB (scalar plus immediate) LD1ROB (scalar plus scalar)
LD1ROD	Contiguous load and replicate four doublewords (scalar plus immediate, scalar plus scalar)	LD1ROD (scalar plus immediate) LD1ROD (scalar plus scalar)
LD1ROH	Contiguous load and replicate 16 halfwords (scalar plus immediate, scalar plus scalar)	LD1ROH (scalar plus immediate) LD1ROH (scalar plus scalar)
LD1ROW	Contiguous load and replicate eight words (scalar plus immediate, scalar plus scalar)	LD1ROW (scalar plus immediate) LD1ROW (scalar plus scalar)
LD1RQB	Contiguous load and replicate 16 bytes (scalar plus immediate, scalar plus scalar)	LD1RQB (scalar plus immediate) LD1RQB (scalar plus scalar)
LD1RQD	Contiguous load and replicate two doublewords (scalar plus immediate, scalar plus scalar)	LD1RQD (scalar plus immediate) LD1RQD (scalar plus scalar)
LD1RQH	Contiguous load and replicate eight halfwords (scalar plus immediate, scalar plus scalar)	LD1RQH (scalar plus immediate) LD1RQH (scalar plus scalar)
LD1RQW	Contiguous load and replicate four words (scalar plus immediate, scalar plus scalar)	LD1RQW (scalar plus immediate) LD1RQW (scalar plus scalar)
LD1RSB	Load and broadcast signed byte to vector	LD1RSB
LD1RSH	Load and broadcast signed halfword to vector	LD1RSH
LD1RSW	Load and broadcast signed word to vector	LD1RSW
LD1RW	Load and broadcast unsigned word to vector	LD1RW

C3.3.5 Unpredicated vector register load/store

The unpredicated vector register load, LDR, and store, STR, instructions transfer a single vector register from or to memory locations that are specified by a scalar base register plus an immediate index value that is in the range -256 to 255, inclusive. The immediate index value defaults to zero if omitted.

The unpredicated vector register load/store instructions have one supporting addressing mode:

- Scalar base plus immediate index.

For the unpredicated vector register load/store SVE instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the current vector register length in bytes.
- The data transfer is performed as a contiguous stream of byte accesses in ascending element order, without endianness conversion.
- When alignment checking is enabled for loads and stores, the base address must be 16-byte aligned.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-71 Unpredicated vector register load/store

Mnemonic	Instruction	See
LDR	Load vector register (vector)	LDR (vector)
STR	Store vector register (vector)	STR (vector)

C3.3.6 Unpredicated predicate register load/store

The unpredicated predicate register load, LDR, and store, STR, instructions transfer a single predicate register from or to memory locations that are specified by a scalar base register plus an immediate index value that is in the range -256 to 255, inclusive. The immediate index value defaults to zero if omitted.

The unpredicated predicate register load/store instructions have one supporting addressing mode:

- Scalar base plus immediate index.

For unpredicated predicate register load/store SVE instructions:

- The immediate index value is a predicate index, not an element index. The immediate index value is multiplied by the current predicate register length, in bytes.
- The data transfer is performed as a contiguous stream of byte accesses, each byte containing 8 consecutive predicate bits, in ascending bit and element order, without endian conversion.
- When alignment checking is enabled for loads and stores, the base address must be 2-byte aligned.

Table C3-72 Unpredicated predicate register load/store

Mnemonic	Instruction	See
LDR	Load predicate register (predicate)	LDR (predicate)
STR	Store predicate register (predicate)	STR (predicate)

C3.3.7 Non-temporal gather/scatter

The non-temporal gather load and scatter store instructions provide a hint to the memory system that the data structure being accessed has a low reuse frequency. The memory system can use the hint to avoid retaining the data or evicting more frequently-used data from the caches.

These instructions support a single addressing mode consisting of 64-bit or 32-bit vector base addresses plus an unscaled 64-bit scalar offset that defaults to the zero register, XZR. Other addressing modes can be constructed using extra instructions.

Table C3-73 Non-temporal gather load and scatter store instructions

Mnemonic	Instruction	See
LDNT1B	Gather load non-temporal unsigned bytes	LDNT1B (vector plus scalar)
LDNT1D	Gather load non-temporal unsigned doublewords	LDNT1D (vector plus scalar)
LDNT1H	Gather load non-temporal unsigned halfwords	LDNT1H (vector plus scalar)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
LDNT1SB	Gather load non-temporal signed bytes	LDNT1SB
LSNT1SH	Gather load non-temporal signed halfwords	LDNT1SH
LSNT1SW	Gather load non-temporal signed words	LDNT1SW
LDNT1W	Gather load non-temporal unsigned words	LDNT1W (vector plus scalar)
STNT1B	Scatter store non-temporal bytes	STNT1B (vector plus scalar)
STNT1D	Scatter store non-temporal doublewords	STNT1D (vector plus scalar)
STNT1H	Scatter store non-temporal halfwords	STNT1H (vector plus scalar)
STNT1W	Scatter store non-temporal words	STNT1W (vector plus scalar)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.4 Loads and stores - SME, SME2, SVE2p1

The following subsections describe the SME, SME2 and SVE2p1 loads and stores:

- Array vector/table load and store.
- Consecutive multi-vector loads and stores.
- SVE2.1 quadword loads and stores.

C3.4.1 Array vector/table load and store

These instructions transfer data between the ZA array vector or the ZT0 registers and memory.

These instructions are performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte.

Table C3-74 SME array vector/table load and store

Mnemonic	Instruction	See
`LDR`	Load ZA array vector	LDR (array vector)
	Load ZT0 register	LDR (table)
`STR`	Store ZA array vector	STR (array vector)
	Store ZT0 register	STR (table)

C3.4.2 Contiguous multi-vector loads and stores

C3.4.2.1 Consecutive multi-vector loads and stores

Multi-vector load and store instructions that transfer data between memory and consecutive SVE vector registers.

Table C3-75 Consecutive multi-vector load and store instructions

Mnemonic	Instruction	See
`LD1`	Contiguous load of bytes to multiple consecutive vectors (immediate index, scalar index)	LD1B (scalar plus immediate, consecutive registers) LD1B (scalar plus scalar, consecutive registers)
	Contiguous load of doublewords to multiple consecutive vectors (immediate index, scalar index)	LD1D (scalar plus immediate, consecutive registers) LD1D (scalar plus scalar, consecutive registers)
	Contiguous load of halfwords to multiple consecutive vectors (immediate index, scalar index)	LD1H (scalar plus immediate, consecutive registers) LD1H (scalar plus scalar, consecutive registers)
	Contiguous load of words to multiple consecutive vectors (immediate index, scalar index)	LD1W (scalar plus immediate, consecutive registers) LD1W (scalar plus scalar, consecutive registers)
`LDNT1`	Contiguous load non-temporal of bytes to multiple consecutive vectors (immediate index, scalar index)	LDNT1B (scalar plus immediate, consecutive registers) LDNT1B (scalar plus scalar, consecutive registers)
	Contiguous load non-temporal of doublewords to multiple consecutive vectors (immediate index, scalar index)	LDNT1D (scalar plus immediate, consecutive registers) LDNT1D (scalar plus scalar, consecutive registers)
	Contiguous load non-temporal of halfwords to multiple consecutive vectors (immediate index, scalar index)	LDNT1H (scalar plus immediate, consecutive registers) LDNT1H (scalar plus scalar, consecutive registers)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Contiguous load non-temporal of words to multiple consecutive vectors (immediate index, scalar index)	LDNT1W (scalar plus immediate, consecutive registers) LDNT1W (scalar plus scalar, consecutive registers)
ST1	Contiguous store of bytes from multiple consecutive vectors (immediate index, scalar index)	ST1B (scalar plus immediate, consecutive registers) ST1B (scalar plus scalar, consecutive registers)
	Contiguous store of doublewords from multiple consecutive vectors (immediate index, scalar index)	ST1D (scalar plus immediate, consecutive registers) ST1D (scalar plus scalar, consecutive registers)
	Contiguous store of halfwords from multiple consecutive vectors (immediate index, scalar index)	ST1H (scalar plus immediate, consecutive registers) ST1H (scalar plus scalar, consecutive registers)
	Contiguous store of words from multiple consecutive vectors (immediate index, scalar index)	ST1W (scalar plus immediate, consecutive registers) ST1W (scalar plus scalar, consecutive registers)
STNT1	Contiguous store non-temporal of bytes from multiple consecutive vectors (immediate index, scalar index)	STNT1B (scalar plus immediate, consecutive registers) STNT1B (scalar plus scalar, consecutive registers)
	Contiguous store non-temporal of doublewords from multiple consecutive vectors (immediate index, scalar index)	STNT1D (scalar plus immediate, consecutive registers) STNT1D (scalar plus scalar, consecutive registers)
	Contiguous store non-temporal of halfwords from multiple consecutive vectors (immediate index, scalar index)	STNT1H (scalar plus immediate, consecutive registers) STNT1H (scalar plus scalar, consecutive registers)
	Contiguous store non-temporal of words from multiple consecutive vectors (immediate index, scalar index)	STNT1W (scalar plus immediate, consecutive registers) STNT1W (scalar plus scalar, consecutive registers)

C3.4.2.2 Strided multi-vector loads and stores

Multi-vector load and store instructions that transfer data between memory and strided numbered vector registers.

Table C3-76 Strided multi-vector load and store instructions

Mnemonic	Instruction	See
LD1	Contiguous load of bytes to multiple strided vectors (immediate index, scalar index)	LD1B (scalar plus immediate, strided registers) LD1B (scalar plus scalar, strided registers)
	Contiguous load of doublewords to multiple strided vectors (immediate index, scalar index)	LD1D (scalar plus immediate, strided registers) LD1D (scalar plus scalar, strided registers)
	Contiguous load of halfwords to multiple strided vectors (immediate index, scalar index)	LD1H (scalar plus immediate, strided registers) LD1H (scalar plus scalar, strided registers)
	Contiguous load of words to multiple strided vectors (immediate index, scalar index)	LD1W (scalar plus immediate, strided registers) LD1W (scalar plus scalar, strided registers)
LDNT1	Contiguous load non-temporal of bytes to multiple strided vectors (immediate index, scalar index)	LDNT1B (scalar plus immediate, strided registers) LDNT1B (scalar plus scalar, strided registers)
	Contiguous load non-temporal of doublewords to multiple strided vectors (immediate index, scalar index)	LDNT1D (scalar plus immediate, strided registers) LDNT1D (scalar plus scalar, strided registers)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Contiguous load non-temporal of halfwords to multiple strided vectors (immediate index, scalar index)	LDNT1H (scalar plus immediate, strided registers) LDNT1H (scalar plus scalar, strided registers)
	Contiguous load non-temporal of words to multiple strided vectors (immediate index, scalar index)	LDNT1W (scalar plus immediate, strided registers) LDNT1W (scalar plus scalar, strided registers)
`ST1`	Contiguous store of bytes from multiple strided vectors (immediate index, scalar index)	ST1B (scalar plus immediate, strided registers) ST1B (scalar plus scalar, strided registers)
	Contiguous store of doublewords from multiple strided vectors (immediate index, scalar index)	ST1D (scalar plus immediate, strided registers) ST1D (scalar plus scalar, strided registers)
	Contiguous store of halfwords from multiple strided vectors (immediate index, scalar index)	ST1H (scalar plus immediate, strided registers) ST1H (scalar plus scalar, strided registers)
	Contiguous store of words from multiple strided vectors (immediate index, scalar index)	ST1W (scalar plus immediate, strided registers) ST1W (scalar plus scalar, strided registers)
`STNT1`	Contiguous store non-temporal of bytes from multiple strided vectors (immediate index, scalar index)	STNT1B (scalar plus immediate, strided registers) STNT1B (scalar plus scalar, strided registers)
	Contiguous store non-temporal of doublewords from multiple strided vectors (immediate index, scalar index)	STNT1D (scalar plus immediate, strided registers) STNT1D (scalar plus scalar, strided registers)
	Contiguous store non-temporal of halfwords from multiple strided vectors (immediate index, scalar index)	STNT1H (scalar plus immediate, strided registers) STNT1H (scalar plus scalar, strided registers)
	Contiguous store non-temporal of words from multiple strided vectors (immediate index, scalar index)	STNT1W (scalar plus immediate, strided registers) STNT1W (scalar plus scalar, strided registers)

C3.4.2.3 Tile slice multi-vector loads and stores

Tile slice load and store instructions transfer bytes between a tile slice and memory.

Table C3-77 Tile slice multi-vector load and store instructions

Mnemonic	Instruction	See
`LD1`	Contiguous load of bytes to 8-bit element ZA tile slice	LD1B (scalar plus scalar, tile slice)
	Contiguous load of doublewords to 64-bit element ZA tile slice	LD1D (scalar plus scalar, tile slice)
	Contiguous load of halfwords to 16-bit element ZA tile slice	LD1H (scalar plus scalar, tile slice)
	Contiguous load of quadwords to 128-bit element ZA tile slice	LD1Q
	Contiguous load of words to 32-bit element ZA tile slice	LD1W (scalar plus scalar, tile slice)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
ST1	Contiguous store of bytes from 8-bit element ZA tile slice	ST1B (scalar plus scalar, tile slice)
	Contiguous store of doublewords from to 64-bit element ZA tile slice	ST1D (scalar plus scalar, tile slice)
	Contiguous store of halfwords from 16-bit element ZA tile slice	ST1H (scalar plus scalar, tile slice)
	Contiguous store of quadwords from 128-bit element ZA tile slice	ST1Q
	Contiguous store of words from 32-bit element ZA tile slice	ST1W (scalar plus scalar, tile slice)

C3.4.3 SVE2.1 quadword loads and stores

C3.4.3.1 Contiguous quadword loads and stores

Contiguous load or store to/from elements of a vector register from/to the memory address generated by a scalar base and an immediate index, which is multiplied and added to the base address, or a scalar index, which is added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

Table C3-78 SME2/SVE2.1 Contiguous quadword load and store instructions

Mnemonic	Instruction	See
LD1D	Contiguous load unsigned doublewords to vector (immediate or scalar index)	LD1D (scalar plus immediate, single register) LD1D (scalar plus scalar, single register)
LD1W	Contiguous load unsigned words to vector (immediate or scalar index)	LD1W (scalar plus immediate, single register) LD1W (scalar plus scalar, single register)
ST1D	Contiguous store doublewords from vector (immediate or scalar index)	ST1D (scalar plus immediate, single register) ST1D (scalar plus scalar, single register)
ST1W	Contiguous store words from vector (immediate or scalar index)	ST1W (scalar plus immediate, single register) ST1W (scalar plus scalar, single register)

C3.4.3.2 Quadword gather and scatter

Gather load or scatter store of quadwords between Active elements of a vector register and memory addresses generated by a vector base plus an unscaled scalar register offset.

Table C3-79 SME2/SVE2.1 Quadword gather and scatter instructions

Mnemonic	Instruction	See
LD1Q	Gather load quadwords	LD1Q
ST1Q	Scatter store quadwords	ST1Q

C3.4.3.3 Quadword structure loads and stores

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-80 SVE2.1 Quadword structure load and store instructions

Mnemonic	Instruction	See
LD	Contiguous load two-quadword structures to two vectors (immediate index, scalar index)	LD2Q (scalar plus immediate) LD2Q (scalar plus scalar)
	Contiguous load three-quadword structures to three vectors (immediate index, scalar index)	LD3Q (scalar plus immediate) LD3Q (scalar plus scalar)
	Contiguous load four-quadword structures to four vectors (immediate index, scalar index)	LD4Q (scalar plus immediate) LD4Q (scalar plus scalar)
ST	Contiguous store two-quadword structures from two vectors (immediate index, scalar index)	ST2Q (scalar plus immediate) ST2Q (scalar plus scalar)
	Contiguous store three-quadword structures from three vectors (immediate index, scalar index)	ST3Q (scalar plus immediate) ST3Q (scalar plus scalar)
	Contiguous store four-quadword structures from four vectors (immediate index, scalar index)	ST4Q (scalar plus immediate) ST4Q (scalar plus scalar)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5 Data processing - immediate

This section describes the instruction groups for data processing with immediate operands. It contains the following subsections:

- Arithmetic (immediate).
- Integer minimum and maximum (immediate)
- Logical (immediate).
- Move (wide immediate).
- Move (immediate).
- PC-relative address calculation.
- Bitfield move.
- Bitfield insert and extract
- Extract register.
- Shift (immediate).
- Sign-extend and Zero-extend.

C3.5.1 Arithmetic (immediate)

The Arithmetic (immediate) instructions accept a 12-bit unsigned immediate value, optionally shifted left by 12 bits.

The Arithmetic (immediate) instructions that do not set Condition flags can read from and write to the current stack pointer. The flag setting instructions can read from the stack pointer, but they cannot write to it.

Table C3-81 shows the Arithmetic instructions with an immediate offset.

Table C3-81 Arithmetic instructions with an immediate

Mnemonic	Instruction	See
`ADD`	Add	ADD (immediate)
`ADDS`	Add and set flags	ADDS (immediate)
`SUB`	Subtract	SUB (immediate)
`SUBS`	Subtract and set flags	SUBS (immediate)
`CMP`	Compare	CMP (immediate)
`CMN`	Compare negative	CMN (immediate)

C3.5.2 Integer minimum and maximum (immediate)

The Integer maximum and minimum (immediate) instructions determine the maximum/minimum of the source register value and immediate.

These instructions are only present when FEAT_CSSC is implemented.

Table C3-82 shows the Integer maximum and minimum (immediate) instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-82 Integer maximum and minimum (immediate) instructions

Mnemonic	Instruction	See
SMAX	Signed Maximum (immediate)	SMAX (immediate)
SMIN	Signed Minimum (immediate)	SMIN (immediate)
UMAX	Unsigned Maximum (immediate)	UMAX (immediate)
UMIN	Unsigned Minimum (immediate)	UMIN (immediate)

C3.5.3 Logical (immediate)

The Logical (immediate) instructions accept a bitmask immediate value that is a 32-bit pattern or a 64-bit pattern viewed as a vector of identical elements of size e = 2, 4, 8, 16, 32 or, 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to (e - 1) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to (e - 1) bits. This mechanism can generate 5334 unique 64-bit patterns as 2667 pairs of pattern and their bitwise inverse.

Note

Values that consist of only zeros or only ones cannot be described in this way.

The Logical (immediate) instructions that do not set the Condition flags can write to the current stack pointer, for example to align the stack pointer in a function prologue.

Note

Apart from ANDS and its TST alias, Logical (immediate) instructions do not set the Condition flags. However, the final results of a bitwise operation can be tested by a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-83 shows the Logical immediate instructions.

Table C3-83 Logical immediate instructions

Mnemonic	Instruction	See
AND	Bitwise AND	AND (immediate)
ANDS	Bitwise AND and set flags	ANDS (immediate)
EOR	Bitwise exclusive OR	EOR (immediate)
ORR	Bitwise inclusive OR	ORR (immediate)
TST	Test bits	TST (immediate)

C3.5.4 Move (wide immediate)

The Move (wide immediate) instructions insert a 16-bit immediate, or inverted immediate, into a 16-bit aligned position in the destination register. The value of the other bits in the destination register depends on the variant used. The optional shift amount can be any multiple of 16 that is smaller than the register size.

Table C3-84 shows the Move (wide immediate) instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-84 Move (wide immediate) instructions

Mnemonic	Instruction	See
MOVZ	Move wide with zero	MOVZ
MOVN	Move wide with NOT	MOVN
MOVK	Move wide with keep	MOVK

C3.5.5 Move (immediate)

The Move (immediate) instructions are aliases for a single MOVZ, MOVN, or ORR (immediate with zero register), instruction to load an immediate value into the destination register. An assembler must permit a signed or unsigned immediate, as long as its binary representation can be generated using one of these instructions, and an assembler error results if the immediate cannot be generated in this way. On disassembly, it is unspecified whether the immediate is output as a signed or an unsigned value.

If there is a choice between the MOVZ, MOVN, and ORR instruction to encode the immediate, then an assembler must prefer MOVZ to MOVN, and MOVZ or MOVN to ORR, to ensure reversability. A disassembler must output ORR (immediate with zero register) MOVZ, and MOVN, as a MOV mnemonic except that the underlying instruction must be used when:

- ORR has an immediate that can be generated by a MOVZ or MOVN instruction.
- A MOVN instruction has an immediate that can be encoded by MOVZ.
- MOVZ #0 or MOVN #0 have a shift amount other than LSL #0.

Table C3-85 shows the Move (immediate) instructions.

Table C3-85 Move (immediate) instructions

Mnemonic	Instruction	See
MOV	Move (inverted wide immediate)	MOV (inverted wide immediate)
	Move (wide immediate)	MOV (wide immediate)
	Move (bitmask immediate)	MOV (bitmask immediate)

C3.5.6 PC-relative address calculation

The ADR instruction adds a signed, 21-bit immediate to the value of the program counter that fetched this instruction, and then writes the result to a general-purpose register. This permits the calculation of any byte address within ±1MB of the current PC.

The ADRP instruction shifts a signed, 21-bit immediate left by 12 bits, adds it to the value of the program counter with the bottom 12 bits cleared to zero, and then writes the result to a general-purpose register. This permits the calculation of the address at a 4KB aligned memory region. In conjunction with an ADD (immediate) instruction, or a load/store instruction with a 12-bit immediate offset, this allows for the calculation of, or access to, any address within ±4GB of the current PC.

Note

The term *page* used in the ADRP description is short-hand for the 4KB memory region, and is not related to the virtual memory translation granule size.

Table C3-86 shows the instructions used for PC-relative address calculations are as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-86 PC-relative address calculation instructions

Mnemonic	Instruction	See
ADRP	Compute address of 4KB page at a PC-relative offset	ADRP
ADR	Compute address of label at a PC-relative offset.	ADR

C3.5.7 Bitfield move

The Bitfield move instructions copy a field of constant width from bit 0 in the source register to a constant bit position in the destination register, or from a constant bit position in the source register to bit 0 in the destination register. The remaining bits in the destination register are set as follows:

- For BFM, the remaining bits are unchanged.
- For UBFM the lower bits, if any, and upper bits, if any, are set to zero.
- For SBFM, the lower bits, if any, are set to zero, and the upper bits, if any, are set to a copy of the most-significant bit in the copied field.

Table C3-87 shows the Bitfield move instructions.

Table C3-87 Bitfield move instructions

Mnemonic	Instruction	See
BFM	Bitfield move	BFM
SBFM	Signed bitfield move	SBFM
UBFM	Unsigned bitfield move (32-bit)	UBFM

C3.5.8 Bitfield insert and extract

The Bitfield insert and extract instructions are implemented as aliases of the Bitfield move instructions. Table C3-88 shows the Bitfield insert and extract aliases.

Table C3-88 Bitfield insert and extract instructions

Mnemonic	Instruction	See
BFC	Bitfield clear	BFC
BFI	Bitfield insert	BFI
BFXIL	Bitfield extract and insert low	BFXIL
SBFIZ	Signed bitfield insert in zero	SBFIZ
SBFX	Signed bitfield extract	SBFX
UBFIZ	Unsigned bitfield insert in zero	UBFIZ
UBFX	Unsigned bitfield extract	UBFX

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See

C3.5.9 Extract register

Depending on the register width of the operands, the Extract register instruction copies a 32-bit or 64-bit field from a constant bit position within a double-width value formed by the concatenation of a pair of source registers to a destination register.

Table C3-89 shows the Extract (immediate) instructions.

Table C3-89 Extract register instructions

Mnemonic	Instruction	See
`EXTR`	Extract register from pair	EXTR

C3.5.10 Shift (immediate)

Shifts and rotates by a constant amount are implemented as aliases of the Bitfield move or Extract register instructions. The shift or rotate amount must be in the range 0 to one less than the register width of the instruction, inclusive.

Table C3-90 shows the aliases that can be used as immediate shift and rotate instructions.

Table C3-90 Aliases for immediate shift and rotate instructions

Mnemonic	Instruction	See
`ASR`	Arithmetic shift right	ASR (immediate)
`LSL`	Logical shift left	LSL (immediate)
`LSR`	Logical shift right	LSR (immediate)
`ROR`	Rotate right	ROR (immediate)

C3.5.11 Sign-extend and Zero-extend

The Sign-extend and Zero-extend instructions are implemented as aliases of the Bitfield move instructions.

Table C3-91 shows the aliases that can be used as zero-extend and sign-extend instructions.

Table C3-91 Zero-extend and sign-extend instructions

Mnemonic	Instruction	See
`SXTB`	Sign-extend byte	SXTB
`SXTH`	Sign-extend halfword	SXTH
`SXTW`	Sign-extend word	SXTW

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
UXTB	Unsigned extend byte	UXTB
UXTH	Unsigned extend halfword	UXTH

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.6 Data Processing - MTE

The following subsections describe the MTE processing instructions:

- Pointer arithmetic instructions.
- Tag generation instructions.

C3.6.1 Pointer arithmetic instructions

Table C3-92 shows the Memory Tagging Extension Pointer Arithmetic instruction.

Table C3-92 MTE Pointer Arithmetic instruction

Mnemonic	Instruction	See
`SUBP`	Subtract pointer	SUBP
`SUBPS`	Subtract pointer, setting flags	SUBPS

C3.6.2 Tag generation instructions

Table C3-93 shows the Memory Tagging Extension Tag generation instructions.

Table C3-93 Tag generation instructions

Mnemonic	Instruction	See
`ADDG`	Add with tag	ADDG
`GMI`	Tag mask insert	GMI
`IRG`	Insert random tag	IRG
`SUBG`	Subtract with tag	SUBG

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7 Data processing - register

This section describes the instruction groups for data processing with all register operands. It contains the following subsections:

- Arithmetic (shifted register).
- Arithmetic (extended register).
- Arithmetic with carry.
- Integer maximum and minimum (register)
- Flag manipulation instructions.
- Logical (shifted register).
- Move (register).
- Absolute value
- Shift (register).
- Multiply and divide.
- CRC32.
- Bit operation.
- Conditional select.
- Conditional comparison.

C3.7.1 Arithmetic (shifted register)

The Arithmetic (shifted register) instructions apply an optional shift operator to the second source register value before performing the arithmetic operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators `LSL`, `ASR`, and `LSR` accept an immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator implies `LSL #0`, which means that there is no shift. A disassembler must not output `LSL #0`. However, a disassembler must output all other shifts by zero.

The current stack pointer, SP or WSP, cannot be used with this class of instructions. See Arithmetic (extended register) for arithmetic instructions that can operate on the current stack pointer.

Table C3-94 shows the Arithmetic (shifted register) instructions.

Table C3-94 Arithmetic (shifted register) instructions

Mnemonic	Instruction	See
`ADD`	Add	ADD (shifted register)
`ADDS`	Add and set flags	ADDS (shifted register)
`SUB`	Subtract	SUB (shifted register)
`SUBS`	Subtract and set flags	SUBS (shifted register)
`CMN`	Compare negative	CMN (shifted register)
`CMP`	Compare	CMP (shifted register)
`NEG`	Negate	NEG (shifted register)
`NEGS`	Negate and set flags	NEGS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7.2 Arithmetic (extended register)

The extended register instructions provide an optional sign-extension or zero-extension of a portion of the second source register value, followed by an optional left shift by a constant amount of 1-4, inclusive.

The extended shift is described by the mandatory extend operator `SXTB`, `SXTH`, `SXTW`, `UXTB`, `UXTH`, or `UXTW` . This is followed by an optional left shift amount. If the shift amount is not specified, the default shift amount is zero. A disassembler must not output a shift amount of zero.

For 64-bit instruction forms, the additional operators `UXTX` and `SXTX` use all 64 bits of the second source register with an optional shift. In that case, Arm recommends `UXTX` as the operator. If and only if at least one register is SP, Arm recommends use of the `LSL` operator name, rather than `UXTX`, and when the shift amount is also zero then both the operator and the shift amount can be omitted. `UXTW` and `SXTW` both use all 32 bits of the second source register with an optional shift. In that case Arm recommends `UXTW` as the operator. If and only if at least one register is WSP, Arm recommends use of the `LSL` operator name, rather than `UXTW`, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

For 32-bit instruction forms, the operators `UXTW` and `SXTW` both use all 32 bits of the second source register with an optional shift. In that case, Arm recommends `UXTW` as the operator. If and only if at least one register is WSP, Arm recommends use of the `LSL` operator name, rather than `UXTW`, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

The non-flag setting variants of the extended register instruction permit the use of the current stack pointer as either the destination register and the first source register. The flag setting variants only permit the stack pointer to be used as the first source register.

In the 64-bit form of these instructions, the final register operand is written as `Wm` for all except the `UXTX`/`LSL` and `SXTX` extend operators. For example:

```
CMP X4, W5, SXTW
ADD X1, X2, W3, UXTB #2
SUB SP, SP, X1              // SUB SP, SP, X1, UXTX #0
```

Table C3-95 shows the Arithmetic (extended register) instructions.

Table C3-95 Arithmetic (extended register) instructions

Mnemonic	Instruction	See
`ADD`	Add	ADD (extended register)
`ADDS`	Add and set flags	ADDS (extended register)
`SUB`	Subtract	SUB (extended register)
`SUBS`	Subtract and set flags	SUBS (extended register)
`CMN`	Compare negative	CMN (extended register)
`CMP`	Compare	CMP (extended register)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7.3 Arithmetic with carry

The Arithmetic with carry instructions accept two source registers, with the carry flag as an additional input to the calculation. They do not support shifting of the second source register.

Table C3-96 shows the Arithmetic with carry instructions

Table C3-96 Arithmetic with carry instructions

Mnemonic	Instruction	See
ADC	Add with carry	ADC
ADCS	Add with carry and set flags	ADCS
SBC	Subtract with carry	SBC
SBCS	Subtract with carry and set flags	SBCS
NGC	Negate with carry	NGC
NGCS	Negate with carry and set flags	NGCS

C3.7.4 Integer maximum and minimum (register)

The Integer maximum and minimum (register) instructions determine the maximum/minimum of the two source register values.

These instructions are only present when FEAT_CSSC is implemented.

Table C3-97 shows the Integer maximum and minimum (register) instructions.

Table C3-97 Integer maximum and minimum (register) instructions

Mnemonic	Instruction	See
SMAX	Signed Maximum (register)	SMAX (register)
SMIN	Signed Minimum (register)	SMIN (register)
UMAX	Unsigned Maximum (register)	UMAX (register)
UMIN	Unsigned Minimum (register)	UMIN (register)

C3.7.5 Flag manipulation instructions

The Flag manipulation instructions set the value of the NZCV condition flags directly.

The instructions SETF8 and SETF16 accept one source register and set the NZV condition flags based on the value of the input register. The instruction RMIF accepts one source register and two immediate values, rotating the first source register using the first immediate value and setting the NZCV condition flags masked by the second immediate value.

The instructions XAFLAG and AXFLAG convert PSTATE condition flags between the FCMP instruction format and an alternative format. See Table C6-1 for more information.

Table C3-98 shows the Flag manipulation instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-98 Flag manipulation instructions

Mnemonic	Instruction	See
AXFLAG	Convert from FCMP comparison format to the alternative format	AXFLAG
CFINV	Invert value of the PSTATE.C bit	CFINV
RMIF	Rotate, mask insert flags	RMIF
SETF8	Evaluation of 8-bit flags	SETF8, SETF16
SETF16	Evaluation of 16-bit flags	SETF8, SETF16
XAFLAG	Convert from alternative format to FCMP comparison format	XAFLAG

C3.7.6 Logical (shifted register)

The Logical (shifted register) instructions apply an optional shift operator to the second source register value before performing the main operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators LSL, ASR, LSR, and ROR accept a constant immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator and amount implies LSL #0, which means that there is no shift. A disassembler must not output LSL #0. However, a disassembler must output all other shifts by zero.

Note

Apart from ANDS, TST, and BICS, the logical instructions do not set the Condition flags, but the final result of a bit operation can usually directly control a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-99 shows the Logical (shifted register) instructions.

Table C3-99 Logical (shifted register) instructions

Mnemonic	Instruction	See
AND	Bitwise AND	AND (shifted register)
ANDS	Bitwise AND and set flags	ANDS (shifted register)
BIC	Bitwise bit clear	BIC (shifted register)
BICS	Bitwise bit clear and set flags	BICS (shifted register)
EON	Bitwise exclusive-OR NOT	EON (shifted register)
EOR	Bitwise exclusive-OR	EOR (shifted register)
ORR	Bitwise inclusive OR	ORR (shifted register)
MVN	Bitwise NOT	MVN
ORN	Bitwise inclusive OR NOT	ORN (shifted register)
TST	Test bits	TST (shifted register)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7.7 Move (register)

The Move (register) instructions are aliases for other data processing instructions. They copy a value from a general-purpose register to another general-purpose register or the current stack pointer, or from the current stack pointer to a general-purpose register.

Table C3-100 MOV register instructions

Mnemonic	Instruction	See
`MOV`	Move register	MOV (register)
	Move register to SP or move SP to register	MOV (to/from SP)

C3.7.8 Absolute value

The Absolute value instruction is only present when FEAT_CSSC is implemented.

Table C3-101 shows the Absolute value instruction.

Table C3-101 Absolute value instruction

Mnemonic	Instruction	See
`ABS`	Absolute value	ABS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7.9 Shift (register)

In the Shift (register) instructions, the shift amount is the positive value in the second source register modulo the register size. The register width of the instruction controls whether the new bits are fed into the result on a right shift or rotate at bit[63] or bit[31].

Table C3-102 shows the Shift (register) instructions.

Table C3-102 Shift (register) instructions

Mnemonic	Instruction	See
`ASRV`	Arithmetic shift right variable	ASRV
`LSLV`	Logical shift left variable	LSLV
`LSRV`	Logical shift right variable	LSRV
`RORV`	Rotate right variable	RORV

However, the Shift (register) instructions have a preferred set of aliases that match the shift immediate aliases described in Shift (immediate).

Table C3-103 shows the aliases for Shift (register) instructions.

Table C3-103 Aliases for Variable shift instructions

Mnemonic	Instruction	See
`ASR`	Arithmetic shift right	ASR (register)
`LSL`	Logical shift left	LSL (register)
`LSR`	Logical shift right	LSR (register)
`ROR`	Rotate right	ROR (register)

C3.7.10 Multiply and divide

This section describes the instructions used for integer multiplication and division. It contains the following subsections:

- Multiply.
- Divide.

C3.7.10.1 Multiply

The Multiply instructions write to a single 32-bit or 64-bit destination register, and are built around the fundamental four operand multiply-add and multiply-subtract operation, together with 32-bit to 64-bit widening variants. A 64-bit to 128-bit widening multiple can be constructed with two instructions, using `SMULH` or `UMULH` to generate the upper 64 bits. Table C3-104 shows the Multiply instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-104 Multiply integer instructions

Mnemonic	Instruction	See
MADD	Multiply-add	MADD
MSUB	Multiply-subtract	MSUB
MNEG	Multiply-negate	MNEG
MUL	Multiply	MUL
SMADDL	Signed multiply-add long	SMADDL
SMSUBL	Signed multiply-subtract long	SMSUBL
SMNEGL	Signed multiply-negate long	SMNEGL
SMULL	Signed multiply long	SMULL
SMULH	Signed multiply high	SMULH
UMADDL	Unsigned multiply-add long	UMADDL
UMSUBL	Unsigned multiply-subtract long	UMSUBL
UMNEGL	Unsigned multiply-negate long	UMNEGL
UMULL	Unsigned multiply long	UMULL
UMULH	Unsigned multiply high	UMULH

C3.7.10.2 Divide

The Divide instructions compute the quotient of a division, rounded towards zero. The remainder can then be computed as (numerator - (quotient × denominator)), using the `MSUB` instruction.

If a signed integer division (`INT_MIN / -1`) is performed where `INT_MIN` is the most negative integer value representable in the selected register size, then the result overflows the signed integer range. No indication of this overflow is produced and the result that is written to the destination register is `INT_MIN`.

A division by zero results in a zero being written to the destination register, without any indication that the division by zero occurred.

Table C3-105 shows the Divide instructions.

Table C3-105 Divide instructions

Mnemonic	Instruction	See
SDIV	Signed divide	SDIV
UDIV	Unsigned divide	UDIV

C3.7.11 CRC32

The `CRC32` instructions operate on the general-purpose register file to update a 32-bit CRC value from an input value comprising 1, 2, 4, or 8 bytes. There are two different classes of `CRC` instructions, `CRC32`, and `CRC32C`, that support two commonly used 32-bit polynomials, known as CRC-32 and CRC-32C.

To fit with common usage, the bit order of the values is reversed as part of the operation.

When bits[19:16] of ID_AA64ISAR0_EL1 are set to `0b0001`, the `CRC` instructions are implemented.

These instructions are optional in an Armv8.0 implementation.

All implementations of Armv8.1 architecture and later are required to implement the `CRC32` instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-106 shows the CRC instructions.

Table C3-106 CRC32 instructions

Mnemonic	Instruction	See
CRC32B	CRC-32 sum from byte	CRC32B, CRC32H, CRC32W, CRC32X
CRC32H	CRC-32 sum from halfword	CRC32B, CRC32H, CRC32W, CRC32X
CRC32W	CRC-32 sum from word	CRC32B, CRC32H, CRC32W, CRC32X
CRC32X	CRC-32 sum from doubleword	CRC32B, CRC32H, CRC32W, CRC32X
CRC32CB	CRC-32C sum from byte	CRC32CB, CRC32CH, CRC32CW, CRC32CX
CRC32CH	CRC-32C sum from halfword	CRC32CB, CRC32CH, CRC32CW, CRC32CX
CRC32CW	CRC-32C sum from word	CRC32CB, CRC32CH, CRC32CW, CRC32CX
CRC32CX	CRC-32C sum from doubleword	CRC32CB, CRC32CH, CRC32CW, CRC32CX

C3.7.12 Bit operation

The CNT and CTZ instructions are only present when FEAT_CSSC is implemented.

Table C3-107 shows the Bit operation instructions.

Table C3-107 Bit operation instructions

Mnemonic	Instruction	See
CLS	Count leading sign bits	CLS
CLZ	Count leading zero bits	CLZ
CNT	Count bits	CNT
CTZ	Count trailing zero bits	CTZ
RBIT	Reverse bit order	RBIT
REV	Reverse bytes in register	REV
REV16	Reverse bytes in halfwords	REV16
REV32	Reverse bytes in words	REV32
REV64	Reverse bytes in register	REV64

C3.7.13 Conditional select

The Conditional select instructions select between the first or second source register, depending on the current state of the Condition flags. When the named condition is true, the first source register is selected and its value is copied without modification to the destination register. When the condition is false the second source register is selected and its value might be optionally inverted, negated, or incremented by one, before writing to the destination register.

Other useful conditional set and conditional unary operations are implemented as aliases of the four Conditional select instructions.

Table C3-108 shows the Conditional select instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-108 Conditional select instructions

Mnemonic	Instruction	See
CSEL	Conditional select	CSEL
CSINC	Conditional select increment	CSINC
CSINV	Conditional select inversion	CSINV
CSNEG	Conditional select negation	CSNEG
CSET	Conditional set	CSET
CSETM	Conditional set mask	CSETM
CINC	Conditional increment	CINC
CINV	Conditional invert	CINV
CNEG	Conditional negate	CNEG

C3.7.14 Conditional comparison

The Conditional comparison instructions provide a conditional select for the NZCV Condition flags, setting the flags to the result of an arithmetic comparison of its two source register values if the named input condition is true, or to an immediate value if the input condition is false. There are register and immediate forms. The immediate form compares the source register to a small 5-bit unsigned value.

Table C3-109 shows the Conditional comparison instructions.

Table C3-109 Conditional comparison instructions

Mnemonic	Instruction	See
CCMN	Conditional compare negative (register)	CCMN (register)
CCMN	Conditional compare negative (immediate)	CCMN (immediate)
CCMP	Conditional compare (register)	CCMP (register)
CCMP	Conditional compare (immediate)	CCMP (immediate)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8 Data processing - SIMD and floating-point

This section describes the instruction groups for data processing with SIMD and floating-point register operands.

Common features of SIMD instructions gives general information about SIMD instructions.

The following subsections describe the scalar floating-point data processing instructions:

- Floating-point move (register).
- Floating-point move (immediate).
- Floating-point conversion.
- Floating-point round to integral value.
- Floating-point multiply-add.
- Floating-point arithmetic (one source).
- Floating-point arithmetic (two sources).
- Floating-point minimum and maximum.
- Floating-point comparison.
- Floating-point conditional select.

The following subsections describe the SIMD data processing instructions:

- SIMD move.
- SIMD arithmetic.
- SIMD compare.
- SIMD widening and narrowing arithmetic.
- SIMD unary arithmetic.
- SIMD by element arithmetic.
- SIMD permute.
- SIMD immediate.
- SIMD shift (immediate).
- SIMD conversions.
- SIMD reduce (across vector lanes).
- SIMD pairwise arithmetic.
- SIMD matrix multiply-accumulate.
- SIMD dot product.
- SIMD table lookup.
- SIMD complex number arithmetic instructions.
- SIMD BFloat16.
- SIMD multiply-accumulate.
- The Cryptographic Extension in AArch64 state.

For information about the floating-point exceptions, see Floating-point exceptions and exception traps.

C3.8.1 Common features of SIMD instructions

A number of SIMD instructions come in three forms:

Wide Indicated by the suffix *W*. The element width of the destination register and the first source operand is double that of the second source operand.

Long Indicated by the suffix *L*. The element width of the destination register is double that of both source operands.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Narrow Indicated by the suffix *N*. The element width of the destination register is half that of both source operands.

In addition, each vector form of the instruction is part of a pair, with a second and upper part suffix of *2*, to identify the variant of the instruction:

- Where a SIMD operation widens or lengthens a 64-bit vector to a 128-bit vector, the instruction provides a second part operation that can extract the source from the upper 64 bits of the source registers.
- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the instruction provides a second-part operation that can pack the result of a second operation into the upper part of the same destination register.

Note

This is referred to as a *lane set specifier*.

C3.8.2 Floating-point move (register)

The Floating-point move (register) instructions copy a scalar floating-point value from one register to another register without performing any conversion.

Some of the Floating-point move (register) instructions overlap with the functionality provided by the Advanced SIMD instructions `DUP`, `INS`, and `UMOV`. However, Arm recommends using the `FMOV` instructions when operating on scalar floating-point data to avoid the creation of scalar floating-point code that depends on the availability of the Advanced SIMD instruction set.

Table C3-110 shows the Floating-point move (register) instructions.

Table C3-110 Floating-point move (register) instructions

Mnemonic	Instruction	See
`FMOV`	Floating-point move register without conversion	FMOV (register)
	Floating-point move to or from general-purpose register without conversion	FMOV (general)

C3.8.3 Floating-point move (immediate)

The Floating-point move (immediate) instructions convert a small constant immediate floating-point value into a half-precision, single-precision, or double-precision scalar floating-point value in a SIMD and floating-point register.

The floating-point constant can be specified either in decimal notation, such as 12.0 or -1.2e1, or as a string beginning with `0x` followed by a hexadecimal representation of the IEEE 754 half-precision, single-precision, or double-precision encoding. Arm recommends that a disassembler uses the decimal notation, provided that this displays the value precisely.

Note

When FEAT_FP16 is not implemented, the only half-precision instructions that are supported are floating-point conversions between half-precision, single-precision, and double-precision.

The floating-point value must be expressible as ($\pm n/16 \times 2^r$), where n is an integer in the range $16 \leq n \leq 31$ and r is an integer in the range of $-3 \leq r \leq 4$, that is a normalized binary floating-point encoding with one sign bit, four bits of fraction, and a 3-bit exponent.

Table C3-111 shows the Floating-point move (immediate) instruction:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-111 Floating-point move (immediate) instruction

Mnemonic	Instruction	See
FMOV	Floating-point move immediate	FMOV (scalar, immediate)

C3.8.4 Floating-point conversion

The following subsections describe the conversion of floating-point values:

- Convert floating-point precision.
- Convert from floating-point single-precision to BFloat16.
- Convert from floating-point to integer or fixed-point.
- Convert from integer or fixed-point to floating-point.

C3.8.4.1 Convert floating-point precision

These instructions convert a floating-point scalar with one precision to a floating-point scalar with a different precision, using the current rounding mode as specified by FPCR.RMode.

Table C3-112 shows the Floating-point precision conversion instruction.

Table C3-112 Floating-point precision conversion instruction

Mnemonic	Instruction	See
FCVT	Floating-point convert precision (scalar)	FCVT

C3.8.4.2 Convert from floating-point single-precision to BFloat16

The BFCVT instruction is provided by FEAT_BF16. This instruction converts a single-precision floating-point input to BFloat16 format, honoring the FPCR rounding mode controls to give a more accurate conversion than simply removing the bottom 16 bits of the input.

Table C3-113 shows this instruction.

Table C3-113 Floating-point single-precision to BFloat16 conversion instruction

Mnemonic	Instruction	See
BFCVT	BFloat16 floating-point convert from single-precision to BFloat16 format (scalar)	BFCVT

C3.8.4.3 Convert from floating-point to integer or fixed-point

Floating-point to integer or fixed-point conversion is supported by the following instruction variants:

- Scalar instruction variant, which converts a floating-point value in a SIMD and floating-point source register to an integer or fixed-point value in a general-purpose destination register.
- Scalar SIMD&FP instruction variant, which converts a floating-point value in a SIMD and floating-point source register to an integer value in a SIMD and floating-point destination register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For a fixed-point value, a final immediate operand indicates that the general-purpose register holds a fixed-point number and `fbits` indicates the number of bits after the binary point. `fbits` is in the range 1- 32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

Table C3-114 shows the floating-point to integer or fixed-point conversion instructions.

Table C3-114 Floating-point to integer or fixed-point conversion instructions

Mnemonic	Instruction	See
`FCVTAS`	Floating-point scalar convert to signed integer, rounding to nearest with ties to away (scalar form)	FCVTAS (scalar)
	Floating-point scalar convert to signed integer, rounding to nearest with ties to away (scalar SIMD&FP form)	FCVTAS (scalar SIMD&FP)
`FCVTAU`	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to away (scalar form)	FCVTAU (scalar)
	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to away (scalar SIMD&FP form)	FCVTAU (scalar SIMD&FP)
`FCVTMS`	Floating-point scalar convert to signed integer, rounding toward minus infinity (scalar form)	FCVTMS (scalar)
	Floating-point scalar convert to signed integer, rounding toward minus infinity (scalar SIMD&FP form)	FCVTMS (scalar SIMD&FP)
`FCVTMU`	Floating-point scalar convert to unsigned integer, rounding toward minus infinity (scalar form)	FCVTMU (scalar)
	Floating-point scalar convert to unsigned integer, rounding toward minus infinity (scalar SIMD&FP form)	FCVTMU (scalar SIMD&FP)
`FCVTNS`	Floating-point scalar convert to signed integer, rounding to nearest with ties to even (scalar form)	FCVTNS (scalar)
	Floating-point scalar convert to signed integer, rounding to nearest with ties to even (scalar SIMD&FP form)	FCVTNS (scalar SIMD&FP)
`FCVTNU`	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to even (scalar form)	FCVTNU (scalar)
	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to even (scalar SIMD&FP form)	FCVTNU (scalar SIMD&FP)
`FCVTPS`	Floating-point scalar convert to signed integer, rounding toward positive infinity (scalar form)	FCVTPS (scalar)
	Floating-point scalar convert to signed integer, rounding toward positive infinity (scalar SIMD&FP form)	FCVTPS (scalar SIMD&FP)
`FCVTPU`	Floating-point scalar convert to unsigned integer, rounding toward positive infinity (scalar form)	FCVTPU (scalar)
	Floating-point scalar convert to unsigned integer, rounding toward positive infinity (scalar SIMD&FP form)	FCVTPU (scalar SIMD&FP)
`FCVTZS`	Floating-point scalar convert to signed integer, rounding toward zero (scalar form)	FCVTZS (scalar, integer)
	Floating-point scalar convert to signed fixed-point, rounding toward zero (scalar form)	FCVTZS (scalar, fixed-point)
	Floating-point scalar convert to signed integer, rounding toward zero (scalar SIMD&FP form)	FCVTZS (scalar SIMD&FP)
`FCVTZU`	Floating-point scalar convert to unsigned integer, rounding toward zero (scalar form)	FCVTZU (scalar, integer)
	Floating-point scalar convert to unsigned fixed-point, rounding toward zero (scalar form)	FCVTZU (scalar, fixed-point)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Floating-point scalar convert to unsigned integer, rounding toward zero (scalar SIMD&FP form)	FCVTZU (scalar SIMD&FP)
FJCVTZS	Floating-point Javascript convert to signed fixed-point, rounding toward zero (scalar form)	FJCVTZS

C3.8.4.4 Convert from integer or fixed-point to floating-point

The integer or fixed-point to floating-point conversion is supported by the following instruction variants:

- Scalar instruction variant, which converts an integer or fixed-point value in a general-purpose register to a floating-point value in a SIMD and floating-point destination register.
- Scalar SIMD&FP variant, which converts an integer value in a SIMD and floating-point source register to a floating-point value in a SIMD and floating-point destination register.

For a fixed-point value, a final immediate operand indicates that the general-purpose register holds a fixed-point number and `fbits` indicates the number of bits after the binary point. `fbits` is in the range 1- 32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

Table C3-115 shows the integer or fixed-point to floating-point conversion instructions.

Table C3-115 Integer or fixed-point to floating-point conversion instructions

Mnemonic	Instruction	See
SCVTF	Signed integer scalar convert to floating-point, using the current rounding mode (scalar form)	SCVTF (scalar, integer)
	Signed fixed-point convert to floating-point, using the current rounding mode (scalar form)	SCVTF (scalar, fixed-point)
	Signed integer scalar convert to floating-point, using the current rounding mode (scalar SIMD&FP form)	SCVTF (scalar SIMD&FP)
UCVTF	Unsigned integer scalar convert to floating-point, using the current rounding mode (scalar form)	UCVTF (scalar, integer)
	Unsigned fixed-point convert to floating-point, using the current rounding mode (scalar form)	UCVTF (scalar, fixed-point)
	Unsigned integer scalar convert to floating-point, using the current rounding mode (scalar SIMD&FP form)	UCVTF (scalar SIMD&FP)

C3.8.5 Floating-point round to integral value

The following subsections describe instructions which round a floating-point number to an integral valued floating-point number in the same format:

- Floating-point round to an integer of the same size as the register
- Floating-point round to 32-bit or 64-bit integer

C3.8.5.1 Floating-point round to an integer of the same size as the register

The following instructions round a floating-point value to an integer floating-point value of the same size.

For these instructions:

- A zero input gives a zero result with the same sign.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An infinite input gives an infinite result with the same sign.
- A NaN is propagated as in normal floating-point arithmetic.

These instructions can cause floating-point exceptions, for more information see Floating-point exceptions and exception traps.

Table C3-116 shows the Floating-point round to integer instructions.

Table C3-116 Floating-point round to integer instructions

Mnemonic	Instruction	See
FRINTA	Floating-point round to integer, to nearest with ties to away	FRINTA (scalar)
FRINTI	Floating-point round to integer, using current rounding mode	FRINTI (scalar)
FRINTM	Floating-point round to integer, toward minus infinity	FRINTM (scalar)
FRINTN	Floating-point round to integer, to nearest with ties to even	FRINTN (scalar)
FRINTP	Floating-point round to integer, toward positive infinity	FRINTP (scalar)
FRINTX	Floating-point round to integer exact, using current rounding mode	FRINTX (scalar)
FRINTZ	Floating-point round to integer, toward zero	FRINTZ (scalar)

C3.8.5.2 Floating-point round to 32-bit or 64-bit integer

The following instructions are present if FEAT_FRINTTS is implemented, The instructions round to a value that fits in a 32-bit integer or a 64-bit integer size, and use either round towards zero or the ambient rounding model.

These instructions can cause floating-point exceptions, for more information see Floating-point exceptions and exception traps.

Table C3-117 shows the Floating-point round to 32-bit or 64-bit integer instructions.

Table C3-117 Floating-point round to integer instructions

Mnemonic	Instruction	See
FRINT32X	Floating-point round to 32-bit integer, using current rounding model	FRINT32X (scalar)
FRINT32Z	Floating-point round to 32-bit integer, toward zero	FRINT32Z (scalar)
FRINT64X	Floating point round to 64-bit integer, using current rounding model	FRINT64X (scalar)
FRINT64Z	Floating point round to 64-bit integer, toward zero	FRINT64Z (scalar)

C3.8.6 Floating-point multiply-add

Table C3-118 shows the Floating-point multiply-add instructions that require three source register operands.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-118 Floating-point multiply-add instructions

Mnemonic	Instruction	See
FMADD	Floating-point scalar fused multiply-add	FMADD
FMSUB	Floating-point scalar fused multiply-subtract	FMSUB
FNMADD	Floating-point scalar negated fused multiply-add	FNMADD
FNMSUB	Floating-point scalar negated fused multiply-subtract	FNMSUB

C3.8.7 Floating-point arithmetic (one source)

Table C3-119 shows the Floating-point arithmetic instructions that require a single source register operand.

Table C3-119 Floating-point arithmetic instructions with one source register

Mnemonic	Instructions	See
FABS	Floating-point scalar absolute value	FABS (scalar)
FNEG	Floating-point scalar negate	FNEG (scalar)
FSQRT	Floating-point scalar square root	FSQRT (scalar)

C3.8.8 Floating-point arithmetic (two sources)

Table C3-120 shows the Floating-point arithmetic instructions that require two source register operands.

Table C3-120 Floating-point arithmetic instructions with two source registers

Mnemonic	Instruction	See
FADD	Floating-point scalar add	FADD (scalar)
FDIV	Floating-point scalar divide	FDIV (scalar)
FMUL	Floating-point scalar multiply	FMUL (scalar)
FNMUL	Floating-point scalar multiply-negate	FNMUL (scalar)
FSUB	Floating-point scalar subtract	FSUB (scalar)

C3.8.9 Floating-point minimum and maximum

The `min(x,y)` and `max(x,y)` operations return a quiet NaN when either *x* or *y* is NaN.

As described in Flushing denormalized numbers to zero, if flushing denormalized inputs to zero is enabled, denormal operands are flushed to zero before comparison, and if the result of the comparison is the flushed value, then a zero value is returned. Where both *x* and *y* are zero, or denormal values flushed to zero, with different signs, then +0.0 is returned by `max()` and -0.0 by `min()`.

The `minNum(x,y)` and `maxNum(x,y)` operations follow the IEEE 754-2008 standard and return the numerical operand when one operand is numerical and the other a quiet NaN. Apart from this additional handling of a single quiet NaN, the result is then identical to `min(x,y)` and `max(x,y)`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-121 shows the Floating-point instructions that can perform floating-point minimum and maximum operations.

Table C3-121 Floating-point minimum and maximum instructions

Mnemonic	Instruction	See
FMAX	Floating-point scalar maximum	FMAX (scalar)
FMAXNM	Floating-point scalar maximum number	FMAXNM (scalar)
FMIN	Floating-point scalar minimum	FMIN (scalar)
FMINNM	Floating-point scalar minimum number	FMINNM (scalar)

C3.8.10 Floating-point comparison

These instructions set the NZCV Condition flags in PSTATE, based on the result of a comparison of two operands. If the floating-point comparisons are *unordered*, where one or both operands are a form of NaN, the C and V bits are set to 1 and the N and Z bits are cleared to 0.

Note

The NZCV flags in the FPSR are associated with AArch32 state. The A64 floating-point comparison instructions do not change the Condition flags in the FPSR.

For the conditional Floating-point comparison instructions, if the condition is TRUE, the flags are updated to the result of the comparison, otherwise the flags are updated to the immediate value that is defined in the instruction encoding.

The quiet compare instructions generate an Invalid Operation floating-point exception if either of the source operands is a signaling NaN. The signaling compare instructions generate an Invalid Operation floating-point exception if either of the source operands is any type of NaN.

Note

If FEAT_FlagM2 is implemented, instructions AXFLAG and XAFLAG convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format. See FEAT_FlagM for more information.

Table C3-122 shows the Floating-point comparison instructions.

Table C3-122 Floating-point comparison instructions

Mnemonic	Instruction	See
FCMP	Floating-point quiet compare	FCMP
FCMPE	Floating-point signaling compare	FCMPE
FCCMP	Floating-point conditional quiet compare	FCCMP
FCCMPE	Floating-point conditional signaling compare	FCCMPE

C3.8.11 Floating-point conditional select

Table C3-123 shows the Floating-point conditional select instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-123 Floating-point conditional select instructions

Mnemonic	Instruction	See
FCSEL	Floating-point scalar conditional select	FCSEL

C3.8.12 SIMD move

The functionality of some data movement instructions overlaps with that provided by the scalar floating-point FMOV instructions described in Floating-point move (register).

Table C3-124 shows the SIMD move instructions.

Table C3-124 SIMD move instructions

Mnemonic	Instruction	See
DUP	Duplicate vector element to vector or scalar	DUP (element)
DUP	Duplicate general-purpose register to vector	DUP (general)
INS[a]	Insert vector element from another vector element	INS (element)
	Insert vector element from general-purpose register	INS (general)
MOV	Move vector element to vector element	MOV (element)
	Move general-purpose register to vector element	MOV (from general)
	Move vector element to scalar	MOV (scalar)
	Move vector element to general-purpose register	MOV (to general)
UMOV	Unsigned move vector element to general-purpose register	UMOV
SMOV	Signed move vector element to general-purpose register	SMOV

a. Disassembles as MOV.

C3.8.13 SIMD arithmetic

Table C3-125 shows the SIMD arithmetic instructions.

Table C3-125 SIMD arithmetic instructions

Mnemonic	Instruction	See
ADD	Add (vector and scalar form)	ADD (vector)
AND	Bitwise AND (vector form)	AND (vector)
BIC	Bitwise bit clear (register) (vector form)	BIC (vector, register)
BIF	Bitwise insert if false (vector form)	BIF
BIT	Bitwise insert if true (vector form)	BIT
BSL	Bitwise select (vector form)	BSL
EOR	Bitwise exclusive-OR (vector form)	EOR (vector)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FABD	Floating-point absolute difference (vector and scalar form)	FABD
FADD	Floating-point add (vector form)	FADD (vector)
FAMAX	Floating-point absolute maximum	FAMAX
FAMIN	Floating-point absolute minimum	FAMIN
FDIV	Floating-point divide (vector form)	FDIV (vector)
FMAX	Floating-point maximum (vector form)	FMAX (vector)
FMAXNM	Floating-point maximum number (vector form)	FMAXNM (vector)
FMIN	Floating-point minimum (vector form)	FMIN (vector)
FMINNM	Floating-point minimum number (vector form)	FMINNM (vector)
FMUL	Floating-point multiply (vector form)	FMUL (vector)
FMULX	Floating-point multiply extended (vector and scalar form)	FMULX
FRECPS	Floating-point reciprocal step (vector and scalar form)	FRECPS
FRSQRTS	Floating-point reciprocal square root step (vector and scalar form)	FRSQRTS
FSCALE	Floating-point adjust exponent by vector	FSCALE
FSUB	Floating-point subtract (vector form)	FSUB (vector)
MLA	Multiply-add (vector form)	MLA (vector)
MLS	Multiply-subtract (vector form)	MLS (vector)
MUL	Multiply (vector form)	MUL (vector)
MOV	Move vector register (vector form)	MOV (vector)
ORN	Bitwise inclusive OR NOT (vector form)	ORN (vector)
ORR	Bitwise inclusive OR (register) (vector form)	ORR (vector, register)
PMUL	Polynomial multiply (vector form)	PMUL
SABA	Signed absolute difference and accumulate (vector form)	SABA
SABD	Signed absolute difference (vector form)	SABD
SHADD	Signed halving add (vector form)	SHADD
SHSUB	Signed halving subtract (vector form)	SHSUB
SMAX	Signed maximum (vector form)	SMAX
SMIN	Signed minimum (vector form)	SMIN
SQADD	Signed saturating add (vector and scalar form)	SQADD
SQDMULH	Signed saturating doubling multiply returning high half (vector and scalar form)	SQDMULH (vector)
SQRSHL	Signed saturating rounding shift left (register) (vector and scalar form)	SQRSHL
SQRDMLAH	Signed saturating rounding doubling multiply accumulate returning high half	SQRDMLAH (vector)
SQRDMLSH	Signed saturating rounding doubling multiply subtract returning high half	SQRDMLSH (vector)
SQRDMULH	Signed saturating rounding doubling multiply returning high half (vector and scalar form)	SQRDMULH (vector)
SQSHL	Signed saturating shift left (register) (vector and scalar form)	SQSHL (register)
SQSUB	Signed saturating subtract (vector and scalar form)	SQSUB
SRHADD	Signed rounding halving add (vector form)	SRHADD
SRSHL	Signed rounding shift left (register) (vector and scalar form)	SRSHL
SSHL	Signed shift left (register) (vector and scalar form)	SSHL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SUB	Subtract (vector and scalar form)	SUB (vector)
UABA	Unsigned absolute difference and accumulate (vector form)	UABA
UABD	Unsigned absolute difference (vector form)	UABD
UHADD	Unsigned halving add (vector form)	UHADD
UHSUB	Unsigned halving subtract (vector form)	UHSUB
UMAX	Unsigned maximum (vector form)	UMAX
UMIN	Unsigned minimum (vector form)	UMIN
UQADD	Unsigned saturating add (vector and scalar form)	UQADD
UQRSHL	Unsigned saturating rounding shift left (register) (vector and scalar form)	UQRSHL
UQSHL	Unsigned saturating shift left (register) (vector and scalar form)	UQSHL (register)
UQSUB	Unsigned saturating subtract (vector and scalar form)	UQSUB
URHADD	Unsigned rounding halving add (vector form)	URHADD
URSHL	Unsigned rounding shift left (register) (vector and scalar form)	URSHL
USHL	Unsigned shift left (register) (vector and scalar form)	USHL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.14 SIMD compare

The SIMD compare instructions compare vector or scalar elements according to the specified condition and set the destination vector element to all ones if the condition holds, or to zero if the condition does not hold.

Note

Some of the comparisons, such as LS, LE, LO, and LT, can be made by reversing the operands and using the opposite comparison, HS, GE, HI, or GT.

Table C3-126 shows that SIMD compare instructions.

Table C3-126 SIMD compare instructions

Mnemonic	Instruction	See
CMEQ	Compare bitwise equal (vector and scalar form)	CMEQ (register)
	Compare bitwise equal to zero (vector and scalar form)	CMEQ (zero)
CMHS	Compare unsigned higher or same (vector and scalar form)	CMHS (register)
CMGE	Compare signed greater than or equal (vector and scalar form)	CMGE (register)
	Compare signed greater than or equal to zero (vector and scalar form)	CMGE (zero)
CMHI	Compare unsigned higher (vector and scalar form)	CMHI (register)
CMGT	Compare signed greater than (vector and scalar form)	CMGT (register)
	Compare signed greater than zero (vector and scalar form)	CMGT (zero)
CMLE	Compare signed less than or equal to zero (vector and scalar form)	CMLE (zero)
CMLT	Compare signed less than zero (vector and scalar form)	CMLT (zero)
CMTST	Compare bitwise test bits nonzero (vector and scalar form)	CMTST
FCMEQ	Floating-point compare equal (vector and scalar form)	FCMEQ (register)
	Floating-point compare equal to zero (vector and scalar form)	FCMEQ (zero)
FCMGE	Floating-point compare greater than or equal (vector and scalar form)	FCMGE (register)
	Floating-point compare greater than or equal to zero (vector and scalar form)	FCMGE (zero)
FCMGT	Floating-point compare greater than (vector and scalar form)	FCMGT (register)
	Floating-point compare greater than zero (vector and scalar form)	FCMGT (zero)
FCMLE	Floating-point compare less than or equal to zero (vector and scalar form)	FCMLE (zero)
FCMLT	Floating-point compare less than zero (vector and scalar form)	FCMLT (zero)
FACGE	Floating-point absolute compare greater than or equal (vector and scalar form)	FACGE
FACGT	Floating-point absolute compare greater than (vector and scalar form)	FACGT

C3.8.15 SIMD widening and narrowing arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-127 shows the SIMD widening and narrowing arithmetic instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-127 SIMD widening and narrowing arithmetic instructions

Mnemonic	Instruction	See
ADDHN, ADDHN2	Add returning high, narrow (vector form)	ADDHN, ADDHN2
PMULL, PMULL2	Polynomial multiply long (vector form)	PMULL, PMULL2
RADDHN, RADDHN2	Rounding add returning high, narrow (vector form)	RADDHN, RADDHN2
RSUBHN, RSUBHN2	Rounding subtract returning high, narrow (vector form)	RSUBHN, RSUBHN2
SABAL, SABAL2	Signed absolute difference and accumulate long (vector form)	SABAL, SABAL2
SABDL, SABDL2	Signed absolute difference long (vector form)	SABDL, SABDL2
SADDL, SADDL2	Signed add long (vector form)	SADDL, SADDL2
SADDW, SADDW2	Signed add wide (vector form)	SADDW, SADDW2
SMLAL, SMLAL2	Signed multiply-add long (vector form)	SMLAL, SMLAL2 (vector)
SMLSL, SMLSL2	Signed multiply-subtract long (vector form)	SMLSL, SMLSL2 (vector)
SMULL, SMULL2	Signed multiply long (vector form)	SMULL, SMULL2 (vector)
SQDMLAL, SQDMLAL2	Signed saturating doubling multiply-add long (vector and scalar form)	SQDMLAL, SQDMLAL2 (vector)
SQDMLSL, SQDMLSL2	Signed saturating doubling multiply-subtract long (vector and scalar form)	SQDMLSL, SQDMLSL2 (vector)
SQDMULL, SQDMULL2	Signed saturating doubling multiply long (vector and scalar form)	SQDMULL, SQDMULL2 (vector)
SSUBL, SSUBL2	Signed subtract long (vector form)	SSUBL, SSUBL2
SSUBW, SSUBW2	Signed subtract wide (vector form)	SSUBW, SSUBW2
SUBHN, SUBHN2	Subtract returning high, narrow (vector form)	SUBHN, SUBHN2
UABAL, UABAL2	Unsigned absolute difference and accumulate long (vector form)	UABAL, UABAL2
UABDL, UABDL2	Unsigned absolute difference long (vector form)	UABDL, UABDL2
UADDL, UADDL2	Unsigned add long (vector form)	UADDL, UADDL2
UADDW, UADDW2	Unsigned add wide (vector form)	UADDW, UADDW2
UMLAL, UMLAL2	Unsigned multiply-add long (vector form)	UMLAL, UMLAL2 (vector)
UMLSL, UMLSL2	Unsigned multiply-subtract long (vector form)	UMLSL, UMLSL2 (vector)
UMULL, UMULL2	Unsigned multiply long (vector form)	UMULL, UMULL2 (vector)
USUBL, USUBL2	Unsigned subtract long (vector form)	USUBL, USUBL2
USUBW, USUBW2	Unsigned subtract wide (vector form)	USUBW, USUBW2

C3.8.16 SIMD unary arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-128 shows the SIMD unary arithmetic instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-128 SIMD unary arithmetic instructions

Mnemonic	Instruction	See
ABS	Absolute value (vector and scalar form)	ABS
CLS	Count leading sign bits (vector form)	CLS (vector)
CLZ	Count leading zero bits (vector form)	CLZ (vector)
CNT	Population count per byte (vector form)	CNT
FABS	Floating-point absolute (vector form)	FABS (vector)
FNEG	Floating-point negate (vector form)	FNEG (vector)
FRECPE	Floating-point reciprocal estimate (vector and scalar form)	FRECPE
FRECPX	Floating-point reciprocal exponent (scalar form)	FRECPX
FRINT32X	Floating-point round to 32-bit integer, using current rounding mode (vector form)	FRINT32X (vector)
FRINT32Z	Floating-point round to 32-bit integer, toward zero (vector form)	FRINT32Z (vector)
FRINT64X	Floating-point round to 64-bit integer, using current rounding mode (vector form)	FRINT64X (vector)
FRINT64Z	Floating-point round to 64-bit integer, toward zero (vector form)	FRINT64Z (vector)
FRINTA	Floating-point round to integer, to nearest with ties to away (vector form)	FRINTA (vector)
FRINTI	Floating-point round to integer, using current rounding mode (vector form)	FRINTI (vector)
FRINTM	Floating-point round to integer, toward minus infinity (vector form)	FRINTM (vector)
FRINTN	Floating-point round to integer, to nearest with ties to even (vector form)	FRINTN (vector)
FRINTP	Floating-point round to integer, toward positive infinity (vector form)	FRINTP (vector)
FRINTX	Floating-point round to integer exact, using current rounding mode (vector form)	FRINTX (vector)
FRINTZ	Floating-point round to integer, toward zero (vector form)	FRINTZ (vector)
FRSQRTE	Floating-point reciprocal square root estimate (vector and scalar form)	FRSQRTE
FSQRT	Floating-point square root (vector form)	FSQRT (vector)
MVN	Bitwise NOT (vector form)	MVN
NEG	Negate (vector and scalar form)	NEG (vector)
NOT	Bitwise NOT (vector form)	NOT
RBIT	Bitwise reverse (vector form)	RBIT (vector)
REV16	Reverse elements in 16-bit halfwords (vector form)	REV16 (vector)
REV32	Reverse elements in 32-bit words (vector form)	REV32 (vector)
REV64	Reverse elements in 64-bit doublewords (vector form)	REV64
SADALP	Signed add and accumulate long pairwise (vector form)	SADALP
SADDLP	Signed add long pairwise (vector form)	SADDLP
SQABS	Signed saturating absolute value (vector and scalar form)	SQABS
SQNEG	Signed saturating negate (vector and scalar form)	SQNEG
SQXTN, SQXTN2	Signed saturating extract narrow (vector form)	SQXTN, SQXTN2
SQXTUN, SQXTUN2	Signed saturating extract unsigned narrow (vector and scalar form)	SQXTUN, SQXTUN2
SUQADD	Signed saturating accumulate of unsigned value (vector and scalar form)	SUQADD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SXTL, SXTL2	Signed extend long	SXTL, SXTL2
UADALP	Unsigned add and accumulate long pairwise (vector form)	UADALP
UADDLP	Unsigned add long pairwise (vector form)	UADDLP
UQXTN, UQXTN2	Unsigned saturating extract narrow (vector form)	UQXTN, UQXTN2
URECPE	Unsigned reciprocal estimate (vector form)	URECPE
URSQRTE	Unsigned reciprocal square root estimate (vector form)	URSQRTE
USQADD	Unsigned saturating accumulate of signed value (vector and scalar form)	USQADD
UXTL, UXTL2	Unsigned extend long	UXTL, UXTL2
XTN, XTN2	Extract narrow (vector form)	XTN, XTN2

C3.8.17 SIMD by element arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-129 shows the SIMD by element arithmetic instructions.

Table C3-129 SIMD by element arithmetic instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add (vector and scalar form)	FMLA (by element)
FMLAL, FMLAL2	Floating-point fused multiply-add long (vector form)	FMLAL, FMLAL2 (by element)
FMLS	Floating-point fused multiply-subtract (vector and scalar form)	FMLS (by element).
FMLSL, FMLSL2	Floating-point fused multiply-subtract long (vector form)	FMLSL, FMLSL2 (by element)
FMUL	Floating-point multiply (vector and scalar form)	FMUL (by element)
FMULX	Floating-point multiply extended (vector and scalar form)	FMULX (by element)
MLA	Multiply-add (vector form)	MLA (by element)
MLS	Multiply-subtract (vector form)	MLS (by element)
MUL	Multiply (vector form)	MUL (by element)
SMLAL, SMLAL2	Signed multiply-add long (vector form)	SMLAL, SMLAL2 (by element)
SMLSL, SMLSL2	Signed multiply-subtract long (vector form)	SMLSL, SMLSL2 (by element)
SMULL, SMULL2	Signed multiply long (vector form)	SMULL, SMULL2 (by element)
SQDMLAL, SQDMLAL2	Signed saturating doubling multiply-add long (vector and scalar form)	SQDMLAL, SQDMLAL2 (by element)
SQDMLSL, SQDMLSL2	Signed saturating doubling multiply-subtract long (vector form)	SQDMLSL, SQDMLSL2 (by element)
SQDMULH	Signed saturating doubling multiply returning high half (vector and scalar form)	SQDMULH (by element)
SQDMULL, SQDMULL2	Signed saturating doubling multiply long (vector and scalar form)	SQDMULL, SQDMULL2 (by element)
SQRDMLAH	Signed saturating rounding doubling multiply accumulate returning high half	SQRDMLSH (by element)
SQRDMLSH	Signed saturating rounding doubling multiply subtract returning high half	SQRDMLSH (vector)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SQRDMULH	Signed saturating rounding doubling multiply returning high half (vector and scalar form)	SQRDMULH (by element)
UMLAL, UMLAL2	Unsigned multiply-add long (vector form)	UMLAL, UMLAL2 (by element)
UMLSL, UMLSL2	Unsigned multiply-subtract long (vector form)	UMLSL, UMLSL2 (by element)
UMULL, UMULL2	Unsigned multiply long (vector form)	UMULL, UMULL2 (by element)

C3.8.18 SIMD permute

Table C3-130 shows the SIMD permute instructions.

Table C3-130 SIMD permute instructions

Mnemonic	Instruction	See
EXT	Extract vector from a pair of vectors	EXT
TRN1	Transpose vectors (primary)	TRN1
TRN2	Transpose vectors (secondary)	TRN2
UZP1	Unzip vectors (primary)	UZP1
UZP2	Unzip vectors (secondary)	UZP2
ZIP1	Zip vectors (primary)	ZIP1
ZIP2	Zip vectors (secondary)	ZIP2

C3.8.19 SIMD immediate

Table C3-131 shows the SIMD immediate instructions.

Table C3-131 SIMD immediate instructions

Mnemonic	Instruction	See
BIC	Bitwise bit clear immediate	BIC (vector, immediate)
FMOV	Floating-point move immediate	FMOV (vector, immediate)
MOVI	Move immediate	MOVI
MVNI	Move inverted immediate	MVNI
ORR	Bitwise inclusive OR immediate	ORR (vector, immediate)

C3.8.20 SIMD shift (immediate)

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-132 shows the SIMD shift immediate instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-132 SIMD shift (immediate) instructions

Mnemonic	Instruction	See
RSHRN, RSHRN2	Rounding shift right narrow immediate (vector form)	RSHRN, RSHRN2
SHL	Shift left immediate (vector and scalar form)	SHL
SHLL, SHLL2	Shift left long (by element size) (vector form)	SHLL, SHLL2
SHRN, SHRN2	Shift right narrow immediate (vector form)	SHRN, SHRN2
SLI	Shift left and insert immediate (vector and scalar form)	SLI
SQRSHRN, SQRSHRN2	Signed saturating rounded shift right narrow immediate (vector and scalar form)	SQRSHRN, SQRSHRN2
SQRSHRUN, SQRSHRUN2	Signed saturating shift right unsigned narrow immediate (vector and scalar form)	SQRSHRUN, SQRSHRUN2
SQSHL	Signed saturating shift left immediate (vector and scalar form)	SQSHL (immediate)
SQSHLU	Signed saturating shift left unsigned immediate (vector and scalar form)	SQSHLU
SQSHRN, SQSHRN2	Signed saturating shift right narrow immediate (vector and scalar form)	SQSHRN, SQSHRN2
SQSHRUN, SQSHRUN2	Signed saturating shift right unsigned narrow immediate (vector and scalar form)	SQSHRUN, SQSHRUN2
SRI	Shift right and insert immediate (vector and scalar form)	SRI
SRSHR	Signed rounding shift right immediate (vector and scalar form)	SRSHR
SRSRA	Signed rounding shift right and accumulate immediate (vector and scalar form)	SRSRA.
SSHLL, SSHLL2	Signed shift left long immediate (vector form)	SSHLL, SSHLL2
SSHR	Signed shift right immediate (vector and scalar form)	SSHR
SSRA	Signed integer shift right and accumulate immediate (vector and scalar form)	SSRA
SXTL, SXTL2	Signed integer extend (vector only)	SXTL, SXTL2
UQRSHRN, UQRSHRN2	Unsigned saturating rounded shift right narrow immediate (vector and scalar form)	UQRSHRN, UQRSHRN2
UQSHL	Unsigned saturating shift left immediate (vector and scalar form)	UQSHL (immediate)
UQSHRN, UQSHRN2	Unsigned saturating shift right narrow immediate (vector and scalar form)	UQSHRN, UQSHRN2
URSHR	Unsigned rounding shift right immediate (vector and scalar form)	URSHR
URSRA	Unsigned integer rounding shift right and accumulate immediate (vector and scalar form)	URSRA
USHLL, USHLL2	Unsigned shift left long immediate (vector form)	USHLL, USHLL2
USHR	Unsigned shift right immediate (vector and scalar form)	USHR
USRA	Unsigned shift right and accumulate immediate (vector and scalar form)	USRA
UXTL, UXTL2	Unsigned integer extend (vector only)	UXTL, UXTL2

C3.8.21 SIMD conversions

C3.8.21.1 Floating-point conversion

Table C3-133 shows the SIMD floating-point conversion instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-133 SIMD floating-point conversion instructions

Mnemonic	Instruction	See
FCVTAS	Floating-point convert to signed integer, rounding to nearest with ties to away (vector and scalar form)	FCVTAS (vector)
FCVTAU	Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector and scalar form)	FCVTAU (vector)
FCVTL, FCVTL2	Floating-point convert to higher precision long (vector form)	FCVTL, FCVTL2
FCVTMS	Floating-point convert to signed integer, rounding toward minus infinity (vector and scalar form)	FCVTMS (vector)
FCVTMU	Floating-point convert to unsigned integer, rounding toward minus infinity (vector and scalar form)	FCVTMU (vector)
FCVTN, FCVTN2	Floating-point convert to lower precision narrow (vector form)	FCVTN, FCVTN2 (double to single-precision, single to half-precision)
FCVTNS	Floating-point convert to signed integer, rounding to nearest with ties to even (vector and scalar form)	FCVTNS (vector)
FCVTNU	Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector and scalar form)	FCVTNU (vector)
FCVTPS	Floating-point convert to signed integer, rounding toward positive infinity (vector and scalar form)	FCVTPS (vector)
FCVTPU	Floating-point convert to unsigned integer, rounding toward positive infinity (vector and scalar form)	FCVTPU (vector)
FCVTXN, FCVTXN2	Floating-point convert to lower precision narrow, rounding to odd (vector and scalar form)	FCVTXN, FCVTXN2
FCVTZS	Floating-point convert to signed integer, rounding toward zero (vector and scalar form)	FCVTZS (vector, integer)
	Floating-point convert to signed fixed-point, rounding toward zero (vector and scalar form)	FCVTZS (vector, fixed-point)
FCVTZU	Floating-point convert to unsigned integer, rounding toward zero (vector and scalar form)	FCVTZU (vector, integer)
	Floating-point convert to unsigned fixed-point, rounding toward zero, (vector and scalar form)	FCVTZU (vector, fixed-point)

C3.8.21.2 Integer conversion

Table C3-134 shows the SIMD integer conversion instructions.

Table C3-134 SIMD integer conversion instructions

Mnemonic	Instruction	See
SCVTF	Signed integer convert to floating-point (vector and scalar form)	SCVTF (vector, integer)
	Signed fixed-point convert to floating-point (vector and scalar form)	SCVTF (vector, fixed-point)
UCVTF	Unsigned integer convert to floating-point (vector and scalar form)	UCVTF (vector, integer)
	Unsigned fixed-point convert to floating-point (vector and scalar form)	UCVTF (vector, fixed-point)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.21.3 FP8 floating-point conversion

Convert each floating-point element of the source vector while scaling the value, and place the results in the corresponding elements of the destination vector.

For instructions that convert other floating-point formats to FP8 formats, each input element is scaled by adding the value specified in FPMR.NSCALE to its unbounded exponent during the conversion.

For instructions that convert FP8 formats to other floating-point formats, each input element is scaled by subtracting the value specified in FPMR.LSCALE or FPMR.LSCALE2 from its unbounded exponent during the conversion.

For more information, see the instruction descriptions.

Table C3-135 shows the SIMD FP8 floating-point conversion instructions.

Table C3-135 SIMD FP8 floating-point conversion instructions

Mnemonic	Instruction	See
`BF1CVTL`, `BF1CVTL2`, `BF2CVTL`, `BF2CVTL2`	8-bit floating-point convert to BFloat16 (vector)	BF1CVTL, BF1CVTL2, BF2CVTL, BF2CVTL2
`F1CVTL`, `F1CVTL2`, `F2CVTL`, `F2CVTL2`	8-bit floating-point convert to half-precision (vector)	F1CVTL, F1CVTL2, F2CVTL, F2CVTL2
`FCVTN`	Half-precision to 8-bit floating-point convert and narrow (vector)	FCVTN (half-precision to 8-bit floating-point)
`FCVTN`, `FCVTN2`	Single-precision to 8-bit floating-point convert and narrow (vector)	FCVTN, FCVTN2 (single-precision to 8-bit floating-point)

C3.8.22 SIMD reduce (across vector lanes)

The SIMD reduce (across vector lanes) instructions perform arithmetic operations horizontally, that is across all lanes of the input vector. They deliver a single scalar result.

Table C3-136 shows the SIMD reduce (across vector lanes) instructions.

Table C3-136 SIMD reduce (across vector lanes) instructions

Mnemonic	Instruction	See
`ADDV`	Add (across vector)	ADDV
`FMAXNMV`	Floating-point maximum number (across vector)	FMAXNMV
`FMAXV`	Floating-point maximum (across vector)	FMAXV
`FMINNMV`	Floating-point minimum number (across vector)	FMINNMV
`FMINV`	Floating-point minimum (across vector)	FMINV
`SADDLV`	Signed add long (across vector)	SADDLV
`SMAXV`	Signed maximum (across vector)	SMAXV
`SMINV`	Signed minimum (across vector)	SMINV

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
UADDLV	Unsigned add long (across vector)	UADDLV
UMAXV	Unsigned maximum (across vector)	UMAXV
UMINV	Unsigned minimum (across vector)	UMINV

C3.8.23 SIMD pairwise arithmetic

The SIMD pairwise arithmetic instructions perform operations on pairs of adjacent elements and deliver a vector result.

Table C3-137 shows the SIMD pairwise arithmetic instructions.

Table C3-137 SIMD pairwise arithmetic instructions

Mnemonic	Instruction	See
ADDP	Add pairwise (vector and scalar form)	ADDP (vector) ADDP (scalar)
FADDP	Floating-point add pairwise (vector and scalar form)	FADDP (vector) FADDP (scalar)
FMAXNMP	Floating-point maximum number pairwise (vector and scalar form)	FMAXNMP (vector) FMAXNMP (scalar)
FMAXP	Floating-point maximum pairwise (vector and scalar form)	FMAXP (vector) FMAXP (scalar)
FMINNMP	Floating-point minimum number pairwise (vector and scalar form)	FMINNMP (vector) FMINNMP (scalar)
FMINP	Floating-point minimum pairwise (vector and scalar form)	FMINP (vector) FMINP (scalar)
SMAXP	Signed maximum pairwise	SMAXP
SMINP	Signed minimum pairwise	SMINP
UMAXP	Unsigned maximum pairwise	UMAXP
UMINP	Unsigned minimum pairwise	UMINP

C3.8.24 SIMD dot product

C3.8.24.1 Integer dot product

FEAT_DotProd provides SIMD instructions that perform the dot product of the four 8-bit subelements of the 32-bit elements of one vector with the four 8-bit subelements of a second vector. It provides two forms of the instructions, each with signed and unsigned versions:

Vector form The dot product is calculated for each element of the first vector with the corresponding element of the second vector.

Indexed form The dot product is calculated for each element of the first vector with the element of the second vector that is indicated by the index argument to the instruction.

Note

That is, a single element from the second vector is used, and the dot product is calculated between each element of the first vector and this single element from the second vector.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-138 shows the SIMD integer dot product instructions.

Table C3-138 SIMD integer dot product instructions

Mnemonic	Instruction	See
SDOT	Signed dot product (vector form)	SDOT (vector)
UDOT	Unsigned dot product (vector form)	UDOT (vector)
SDOT	Signed dot product (indexed form)	SDOT (by element)
UDOT	Unsigned dot product (indexed form)	UDOT (by element)
USDOT	Mixed sign integer dot product (vector form)[a]	USDOT (vector)
	Mixed sign integer dot product by indexed quadruplet[a]	USDOT (by element)
SUDOT	Mixed sign integer dot product by indexed quadruplet[a]	SUDOT (by element)

a. This instruction is supported when FEAT_I8MM is implemented.

C3.8.24.2 FP8 floating-point dot product

The FP8 floating-point dot product instructions accumulate the dot product of two or four adjacent FP8 source elements into half-precision or single-precision destination elements. They generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding.

For these instructions, the unbounded intermediate sum-of-products is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-139 shows these instructions.

Table C3-139 SIMD FP8 floating-point dot product instructions

Mnemonic	Instruction	See
FDOT	8-bit floating-point dot product to half-precision (vector, by element)	FDOT (8-bit floating-point to half-precision, by element)
	8-bit floating-point dot product to half-precision (vector)	FDOT (8-bit floating-point to half-precision, vector)
	8-bit floating-point dot product to single-precision (vector, by element)	FDOT (8-bit floating-point to single-precision, by element)
	8-bit floating-point dot product to single-precision (vector)	FDOT (8-bit floating-point to single-precision, vector)

C3.8.25 SIMD matrix multiply-accumulate

C3.8.25.1 Integer matrix multiply-accumulate

The integer matrix multiply-accumulate instructions are implemented by FEAT_I8MM.

The source and destination vectors of this instruction are organized as follows:

- In the first source vector, a 2x8 matrix of 8-bit integers in row-by-row order.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In the second source vector, an 8x2 matrix of 8-bit integers in column-by-column order.
- In the destination vector, a 2x2 matrix of 32-bit integers in row-by-row order.

One matrix multiplication is performed per vector and accumulated into the destination vector.

Table C3-140 shows these instructions.

Table C3-140 SIMD integer matrix multiply-accumulate instructions

Mnemonic	Instruction	See
`SMMLA`	Widening signed 8-bit to 32-bit integer matrix multiply-accumulate	SMMLA (vector)
`UMMLA`	Widening unsigned 8-bit to 32-bit integer matrix multiply-accumulate	UMMLA (vector)
`USMMLA`	Widening mixed sign 8-bit to 32-bit integer matrix multiply-accumulate	USMMLA (vector)

C3.8.25.2 FP8 matrix multiply-accumulate

The FP8 `FMMLA` (widening, 8-bit floating-point to single-precision) instruction is implemented by FEAT_F8F32MM. The source and destination vectors of this instruction are organized as follows:

- In the first source vector, a 2x8 matrix of 8-bit floating-point values in row-by-row order.
- In the second source vector, an 8x2 matrix of 8-bit floating-point values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per vector and accumulated into the destination vector.

The FP8 `FMMLA` (widening, 8-bit floating-point to half-precision) instruction is implemented by FEAT_F8F16MM. This instruction delimits each source and destination vector into 64-bit segments. The 64-bit segments are organized as follows:

- In the first source vector, a 2x4 matrix of 8-bit floating-point values in row-by-row order.
- In the second source vector, a 4x2 matrix of 8-bit floating-point values in column-by-column order.
- In the destination vector, a 2x2 matrix of half-precision values in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

These instructions generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding.

For these instructions, the intermediate sum of products is downscaled by subtracting the value in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-141 SIMD FP8 matrix multiply-accumulate instructions

Mnemonic	Instruction	See
`FMMLA`	8-bit floating-point matrix multiply-accumulate into 2x2 single-precision matrix	FMMLA (widening, 8-bit floating-point to single-precision)
`FMMLA`	8-bit floating-point matrix multiply-accumulate into 2x2 half-precision matrix	FMMLA (widening, 8-bit floating-point to half-precision)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.26 SIMD table lookup

C3.8.26.1 Table vector lookup

Table C3-142 shows the SIMD table vector lookup instructions.

Table C3-142 SIMD table vector lookup instructions

Mnemonic	Instruction	See
TBL	Table vector lookup	TBL
TBX	Table vector lookup extension	TBX

C3.8.26.2 Lookup table using 2-bit or 4-bit indices

Copy indexed elements from the first source vector(s) to the destination vector, using packed indices from a portion of the second source vector.

Table C3-143 shows the SIMD lookup table instructions.

Table C3-143 SIMD lookup table using 2-bit or 4-bit indices instructions

Mnemonic	Instruction	See
LUTI2	Lookup table using 2-bit indices	LUTI2
LUTI4	Lookup table using 4-bit indices	LUTI4

C3.8.27 SIMD complex number arithmetic instructions

FEAT_FCMA provides SIMD instructions that perform arithmetic on complex numbers held in element pairs in vector registers, where the less significant element of the pair contains the real component and the more significant element contains the imaginary component.

These instructions provide double-precision and single-precision versions. If FEAT_FP16 is implemented they also provide half-precision versions, otherwise the half-precision encodings are UNDEFINED.

Table C3-144 shows the FEAT_FCMA SIMD instructions.

Table C3-144 SIMD complex number arithmetic instructions

Mnemonic	Instruction	See
FCADD	Floating-point complex add	FCADD
FCMLA	Floating-point complex multiply accumulate (vector form)	FCMLA
FCMLA	Floating-point complex multiply accumulate (indexed form)	FCMLA (by element)

A pair of FCMLA instructions can be used to perform a complex number multiplication. This is demonstrated in Complex multiplication.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.28 SIMD multiply-accumulate

C3.8.28.1 Floating-point fused multiply-accumulate

The SIMD floating-point fused multiply-accumulate instructions multiply the vector elements in the first source SIMD&FP register by the corresponding value in the second source SIMD&FP register, and accumulate or subtract the product in the vector elements of the destination SIMD&FP register.

Table C3-145 shows these instructions.

Table C3-145 SIMD floating-point fused multiply-accumulate instructions

Mnemonic	Instruction	See
`FMLA`	Floating-point fused multiply-add (vector form)	FMLA (vector)
`FMLAL`, `FMLAL2`	Floating-point fused multiply-add long (vector form)	FMLAL, FMLAL2 (vector)
`FMLS`	Floating-point fused multiply-subtract (vector form)	FMLS (vector)
`FMLSL`, `FMLSL2`	Floating-point fused multiply-subtract long (vector form)	FMLSL, FMLSL2 (vector)

C3.8.28.2 FP8 floating-point widening multiply-accumulate

The FP8 floating-point multiply-accumulate instructions widen the 8-bit floating-point elements in the source vectors to half-precision or single-precision format and multiply the corresponding elements.

They generate a fused, downscaled product and accumulate into the corresponding elements of the destination vector, without intermediate rounding.

For these instructions, the unbounded intermediate product is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-146 shows these instructions.

Table C3-146 SIMD FP8 floating-point widening multiply-accumulate instructions

Mnemonic	Instruction	See
`FMLALB, FMLALT`	8-bit floating-point multiply-add long to half-precision (vector, by element)	FMLALB, FMLALT (by element)
	8-bit floating-point multiply-add long to half-precision (vector)	FMLALB, FMLALT (vector)
`FMLALLBB`, `FMLALLBT`, `FMLALLTB`, `FMLALLTT`	8-bit floating-point multiply-add long-long to single-precision (vector, by element)	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element)
	8-bit floating-point multiply-add long-long to single-precision (vector)	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector)

C3.8.29 SIMD BFloat16

If FEAT_BF16 is implemented, the instructions in this section are available.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.29.1 SIMD BFloat16 floating-point multiply-add

The BFloat16 floating-point multiply-add instructions perform an implicit conversion of the bottom (even-numbered) or top (odd-numbered) BFloat16 source elements to IEEE 754 single-precision floating-point format before performing a fused multiply-add without intermediate rounding to the overlapping single-precision destination element. These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

Table C3-147 SIMD BFloat16 floating-point multiply-add instructions

Mnemonic	Instruction	See
`BFMLALB`	BFloat16 floating-point widening multiply-add long bottom (vector and indexed forms)	BFMLALB, BFMLALT (vector) BFMLALB, BFMLALT (by element)
`BFMLALT`	BFloat16 floating-point widening multiply-add long top (vector and index forms)	BFMLALB, BFMLALT (vector) BFMLALB, BFMLALT (by element)

C3.8.29.2 SIMD BFloat16 floating-point dot product

The BFloat16 floating-point dot product instruction, `BFDOT`, performs an implicit conversion of vectors of BFloat16 input values to IEEE 754 single-precision floating-point format.

The BFloat16 dot product instructions delimit their source vectors into pairs of BFloat16 elements.

Table C3-148 SIMD BFloat16 floating-point dot product instructions

Mnemonic	Instruction	See
`BFDOT`	BFloat16 floating-point dot product (vector and indexed forms)	BFDOT (vector) BFDOT (by element)

C3.8.29.3 SIMD BFloat16 floating-point matrix multiply-accumulate

The BFloat16 floating-point matrix multiply-accumulate instruction, `BFMMLA` is implemented by FEAT_BF16.

The source and destination vectors of this instruction are organized as follows:

- In the first source vector, a 2x4 matrix of BFloat16 values in row-by-row order.
- In the second source vector, a 4x2 matrix of BFloat16 values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per source vector and accumulated into the corresponding destination vector. This corresponds to accumulating two 2-way BFloat16 widening dot products into each single-precision destination element, following the numeric behaviors described for `BFDOT` instruction in SIMD BFloat16 floating-point dot product.

Table C3-149 SIMD BFloat16 floating-point matrix multiply instructions

Mnemonic	Instruction	See
`BFMMLA`	BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix	BFMMLA

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.29.4 SIMD BFloat16 floating-point convert

The BFloat16 floating-point convert instructions perform accurately rounded down-conversion of single-precision source vector elements to BFloat16 format.

Table C3-150 SIMD BFloat16 floating-point convert instructions

Mnemonic	Instruction	See
BFCVTN, BFCVTN2	Floating-point convert from single-precision to BFloat16 format (vector form)	BFCVTN, BFCVTN2

C3.8.30 The Cryptographic Extension in AArch64 state

C3.8.30.1 The Armv8.0 Cryptographic Extension

The instructions provided by the OPTIONAL Armv8.0 Cryptographic Extension use the SIMD and floating-point register file. For more information about the functions they provide, see:

- *Advanced Encryption Standard (AES)*.
- *Secure Hash Algorithm (SHA)*.
- *The Galois/Counter Mode of Operation*.

Table C3-151 shows the Armv8.0 Cryptographic Extension instructions.

Table C3-151 Cryptographic Extension instructions

Mnemonic	Instruction	See
AESD	AES single round decryption	AESD
AESE	AES single round encryption	AESE
AESIMC	AES inverse mix columns	AESIMC
AESMC	AES mix columns	AESMC
PMULL	Polynomial multiply long	PMULL, PMULL2 [a]
SHA1C	SHA1 hash update (choose)	SHA1C
SHA1H	SHA1 fixed rotate	SHA1H
SHA1M	SHA1 hash update (majority)	SHA1M
SHA1P	SHA1 hash update (parity)	SHA1P
SHA1SU0	SHA1 schedule update 0	SHA1SU0
SHA1SU1	SHA1 schedule update 1	SHA1SU1
SHA256H	SHA256 hash update, part 1	SHA256H
SHA256H2	SHA256 hash update, part 2	SHA256H2
SHA256SU0	SHA256 schedule update 0	SHA256SU0
SHA256SU1	SHA256 schedule update 1	SHA256SU1

a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.30.2 Armv8.2 extensions to the Cryptographic Extension

Armv8.2 supports the following OPTIONAL extensions to the Cryptographic Extension:

- FEAT_SHA512, SHA2-512 functionality.
- FEAT_SHA3, SHA3 functionality.
- FEAT_SM3, SM3 functionality.
- FEAT_SM4, SM4 functionality.

C3.8.30.2.1 FEAT_SHA512, SHA2-512 functionality

FEAT_SHA512 provides instructions to accelerate the SHA-2 hash algorithm using a digest that is larger than 256 bits. The relevant standards are SHA-384, SHA-512, SHA-512|224 and SHA-512|256. These are all based on the SHA-512 computation, and therefore this set of instructions is described as the *SHA512 instructions*.

Implementation of FEAT_SHA512 requires the implementation of the SHA1 and SHA2-256 instructions from the Armv8.0 Cryptographic Extension.

Note

Implementation of FEAT_SHA512 does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the `PMULL` instructions, from the Armv8.0 Cryptographic Extension.

When FEAT_SHA512 is implemented, the value of ID_AA64ISAR0_EL1.SHA2 is `0b0010`, indicating support for the SHA512 instructions.

Table C3-152 shows the FEAT_SHA512 instructions:

Table C3-152 FEAT_SHA512 instructions

Mnemonic	Instruction	See
`SHA512H`	SHA512 Hash update part 1	SHA512H
`SHA512H2`	SHA512 Hash update part 2	SHA512H2
`SHA512SU0`	SHA512 Schedule Update 0	SHA512SU0
`SHA512SU1`	SHA512 Schedule Update 1	SHA512SU1

Use of the SHA512 instructions shows an example of the use of these instructions to calculate a SHA512 hash iteration. This example code is not part of the architectural definition of these instructions.

C3.8.30.2.2 FEAT_SHA3, SHA3 functionality

FEAT_SHA3 provides instructions to accelerate the SHA-3 hash algorithm. This set of instructions is described as the *SHA3 instructions.*

Note

Implementation of FEAT_SHA3 does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the `PMULL` instructions, from the Armv8.0 Cryptographic Extension.

When FEAT_SHA3 is implemented, the value of ID_AA64ISAR0_EL1.SHA3 is `0b0001`, indicating support for the SHA3 instructions.

Table C3-153 shows the FEAT_SHA3 instructions. The SHA-3 hash algorithm is based on a running digest of 1600 bytes, arranged as a five by five array of 64-bit registers. The Arm acceleration of these instructions is based on mapping the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is performed on these registers as part of a round of the SHA-3 hash calculation.

The SIMD nature of the vector registers means the acceleration can compute two parallel SHA3 hash calculations, where one calculation is performed using the zeroth 64-bit element of each vector, and the other calculation is performed using the first 64-bit element of each vector.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To provide acceleration where the SIMD calculation is not required, the instructions provide variants that operate only on the zeroth 64-bit elements. These are provided as a power optimization.

Table C3-153 FEAT_SHA3 instructions

Mnemonic	Instruction	See
BCAX	Bit Clear and Exclusive OR	BCAX
EOR3	Three-way Exclusive OR	EOR3
RAX1	Rotate and Exclusive OR	RAX1
XAR	Exclusive OR and Rotate	XAR

Use of the SHA3 instructions shows an example of the use of these instructions to calculate the combined theta, phi, rho and chi operations of a SHA3 iteration. This example code is not part of the architectural definition of these instructions.

C3.8.30.2.3 FEAT_SM3, SM3 functionality

FEAT_SM3 provides instructions to accelerate the SM3 hash algorithm, the standard Chinese hash algorithm. These are described as the *SM3 instructions*.

FEAT_SM3 can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of FEAT_SHA512.

Note

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the FEAT_SM3 functionality.

When FEAT_SM3 is implemented, the value of ID_AA64ISAR0_EL1.SM3 is `0b0001`, indicating support for the SM3 instructions.

Table C3-154 shows the FEAT_SM3 instructions. The SM3 algorithm computes a digest of 256 bits, which can be held in two vector registers. The SM3 instructions include instructions to accelerate the computation of the hash and the schedule update.

Note

The SM3 instruction names refer to intermediate variables defined as part of the *SM3 Cryptographic Hash Algorithm* specification.

Table C3-154 FEAT_SM3 instructions

Mnemonic	Instruction	See
SM3SS1	SM3 SS1 calculation	SM3SS1
SM3TT1A	SM3 TT1 calculation, part A	SM3TT1A
SM3TT1B	SM3 TT1 calculation, part B	SM3TT1B
SM3TT2A	SM3 TT2 calculation, part A	SM3TT2A
SM3TT2B	SM3 TT2 calculation, part B	SM3TT2B
SM3PARTW1	SM3 PARTW calculation, part 1	SM3PARTW1
SM3PARTW2	SM3 PARTW calculation, part 1	SM3PARTW2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Use of the SM3 instructions shows an example of the use of these instructions to generate an SM3 hash. This example code is not part of the architectural definition of these instructions.

C3.8.30.2.4 FEAT_SM4, SM4 functionality

FEAT_SM4 provides instruction to accelerate the SM4 encryption algorithm, the standard Chinese encryption algorithm. This set of instructions is described as the *SM4 instructions*.

FEAT_SM4 can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of FEAT_SHA3.

Note

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the FEAT_SM4 functionality.

When FEAT_SM4 is implemented, the value of ID_AA64ISAR0_EL1.SM4 is `0b0001`, indicating support for the SM4 instructions.

Table C3-155 shows the FEAT_SM4 instructions. The SM4 algorithm is 128-bit wide block cipher. The `SM4E` instruction accelerates a single round of encryption or decryption, and the `SM4EKEY` instruction accelerates a single round of key generation:

Table C3-155 FEAT_SM4 instructions

Mnemonic	Instruction	See
`SM4E`	SM4 Encrypt	SM4E
`SM4EKEY`	SM4 Key	SM4EKEY

Use of the SM4 instructions shows an example of the use of these instructions to perform SM4 encryption and decryption. This example code is not part of the architectural definition of these instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.9 Data processing - SVE

The following subsections describe the SVE processing instructions:

- SVE Vector integer operations
- SVE Vector address calculation
- SVE Bitwise logical operations
- SVE Bitwise shift, reverse, and count
- SVE Vector floating-point operations
- Predicate operations
- Loop control
- SVE Move operations
- Index vector generation
- Move prefix
- Reduction operations

C3.9.1 SVE Vector integer operations

C3.9.1.1 SVE Integer arithmetic

For binary operations, the Integer arithmetic instructions perform arithmetic operations on a source vector containing integer element values, and a second source vector of either integer element values or an immediate value.

For ternary operations, these instructions perform arithmetic operations on a source vector containing integer element values, a second source vector of either integer element values or an immediate value, and a third source vector containing integer element values.

Table C3-156 Integer arithmetic instructions

Mnemonic	Instruction	See
ABS	Absolute value	ABS
ADD	Add vectors (predicated)	ADD (vectors, predicated)
	Add vectors (unpredicated)	ADD (vectors, unpredicated)
	Add immediate	ADD (immediate)
CNOT	Logically invert Boolean condition	CNOT
MAD	Multiply-add, writing to the multiplicand register	MAD
MLA	Multiply-add, writing to the addend register	MLA (indexed) MLA (vectors)
MLS	Multiply-subtract, writing to the addend register	MLS (indexed) MLS (vectors)
MSB	Multiply-subtract, writing to the multiplicand register	MSB
MUL	Multiply by immediate	MUL (immediate)
	Multiply vectors	MUL (vectors, predicated) MUL (vectors, unpredicated)
NEG	Negate	NEG
SABD	Signed absolute difference	SABD
SDIV	Signed divide	SDIV

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SDIVR	Signed reversed divide	SDIVR
SMAX	Signed maximum with immediate	SMAX (immediate)
	Signed maximum vectors	SMAX (vectors)
SMIN	Signed minimum with immediate	SMIN (immediate)
	Signed minimum vectors	SMIN (vectors)
SMULH	Signed multiply returning high half	SMULH (predicated) SMULH (unpredicated)
SQADD	Signed saturating add immediate	SQADD (immediate)
	Signed saturating add vectors	SQADD (vectors, predicated) SQADD (vectors, unpredicated)
SQSUB	Signed saturating subtract immediate	SQSUB (immediate)
	Signed saturating subtract vectors	SQSUB (vectors, predicated) SQSUB (vectors, unpredicated)
SUB	Subtract immediate	SUB (immediate)
	Subtract vectors (predicated)	SUB (vectors, predicated)
	Subtract vectors (unpredicated)	SUB (vectors, unpredicated)
SUBR	Reversed subtract from immediate	SUBR (immediate)
	Reversed subtract vectors	SUBR (vectors)
SXTB	Signed byte extend	SXTB, SXTH, SXTW
SXTH	Signed halfword extend	SXTB, SXTH, SXTW
SXTW	Signed word extend	SXTB, SXTH, SXTW
UABD	Unsigned absolute difference	UABD
UDIV	Unsigned divide	UDIV
UDIVR	Unsigned reversed divide	UDIVR
UMAX	Unsigned maximum with immediate	UMAX (immediate)
	Unsigned maximum vectors	UMAX (vectors)
UMIN	Unsigned minimum with immediate	UMIN (immediate)
	Unsigned minimum vectors	UMIN (vectors)
UMULH	Unsigned multiply returning high half	UMULH (predicated) UMULH (unpredicated)
UQADD	Unsigned saturating add immediate	UQADD (immediate)
	Unsigned saturating add vectors	UQADD (vectors, predicated) UQADD (vectors, unpredicated)
UQSUB	Unsigned saturating subtract immediate	UQSUB (immediate)
	Unsigned saturating subtract vectors	UQSUB (vectors, predicated) UQSUB (vectors, unpredicated)
UXTB	Unsigned byte extend	UXTB, UXTH, UXTW
UXTH	Unsigned halfword extend	UXTB, UXTH, UXTW
UXTW	Unsigned word extend	UXTB, UXTH, UXTW

C3.9.1.2 SVE Integer dot product

The Integer dot product instructions delimit the source vectors into groups of four 8-bit or 16-bit integer elements. Within each group of four elements, the elements in the first source vector are multiplied by the corresponding elements

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

in the second source vector. The resulting widened products are summed and added to the 32-bit or 64-bit element of the accumulator and destination vector that aligns with the group of four elements in the first source vector.

The indexed forms of these instructions specify a single, numbered, group of four elements within each 128-bit segment of the second source vector as the multiplier for all the groups of four elements within the corresponding 128-bit segment of the first source vector.

Table C3-157 Integer dot product instructions

Mnemonic	Instruction	See
SDOT	Signed integer indexed dot product	SDOT (2-way, indexed)
	Signed integer dot product	SDOT (2-way, vectors)
SDOT	Signed dot product by vector	SDOT (4-way, vectors)
	Signed dot product by indexed elements	SDOT (4-way, indexed)
SUDOT	Signed by unsigned integer dot product by indexed elements	SUDOT
UDOT	Unsigned integer indexed dot product	UDOT (2-way, indexed)
	Unsigned integer dot product	UDOT (2-way, vectors)
UDOT	Unsigned dot product by vector	UDOT (4-way, vectors)
	Unsigned dot product by indexed elements	UDOT (4-way, indexed)
USDOT	Unsigned by signed integer dot product	USDOT (vectors)
	Unsigned by signed integer dot product by indexed elements	USDOT (indexed)

C3.9.1.3 SVE Integer matrix multiply-accumulate

The Integer matrix multiply-accumulate instructions are implemented by FEAT_I8MM.

The matrix multiply-accumulate instructions delimit source and destination vectors into 128-bit segments. The 128-bit segments are organized as follows:

- In the first source vector, a 2x8 matrix of 8-bit integers in row-by-row order.
- In the second source vector, an 8x2 matrix of 8-bit integers in column-by-column order.
- In the destination vector, a 2x2 matrix of 32-bit integers in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

Table C3-158 Integer matrix multiply operations

Mnemonic	Instruction	See
SMMLA	Widening signed 8-bit integer matrix multiply-accumulate into 2x2 matrix	SMMLA
UMMLA	Widening unsigned 8-bit integer matrix multiply-accumulate into 2x2 matrix	UMMLA
USMMLA	Widening mixed sign 8-bit integer matrix multiply-accumulate into 2x2 matrix	USMMLA

C3.9.1.4 SVE Integer comparison

The Integer comparison instructions compare *Active elements* in the first source vector with the corresponding elements in a second vector or with an immediate value. The Boolean result of each comparison is placed in the corresponding element of the destination predicate. *Inactive elements* in the destination predicate register are set to FALSE. All integer comparisons set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The wide element variants of the compare instructions allow a packed vector of narrower elements to be compared with wider 64-bit elements. These instructions treat the second source vector as having a fixed 64-bit doubleword element size and compare each narrow element of the first source vector with the corresponding vertically-aligned wide element of the second source vector. For example, if the first source vector contained 8-bit byte elements, then 8-bit element[0] to element[7] of the first source vector are compared with 64-bit element[0] of the second source vector, 8-bit element[8] to element[15] with 64-bit element[1], and so on. All 64 bits of the wide elements are significant for the comparison, with the narrow elements being sign-extended or zero-extended to 64 bits as appropriate for the type of comparison.

Table C3-159 Integer comparison instructions

Mnemonic	Instruction	See
CMPEQ	Compare signed equal to immediate	CMP<cc> (immediate) Equal
	Compare signed equal to wide elements	CMP<cc> (wide elements) Equal
	Compare signed equal to vector	CMP<cc> (vectors) Equal
CMPGE	Compare signed greater than or equal to immediate	CMP<cc> (immediate) Greater than or equal
	Compare signed greater than or equal to wide elements	CMP<cc> (wide elements) Greater than or equal
	Compare signed greater than or equal to vector	CMP<cc> (vectors) Greater than or equal
CMPGT	Compare signed greater than immediate	CMP<cc> (immediate) Greater than
	Compare signed greater than wide elements	CMP<cc> (wide elements) Greater than
	Compare signed greater than vector	CMP<cc> (vectors) Greater than
CMPHI	Compare unsigned higher than immediate	CMP<cc> (immediate) Higher
	Compare unsigned higher than wide elements	CMP<cc> (wide elements) Higher
	Compare unsigned higher than vector	CMP<cc> (vectors) Higher
CMPHS	Compare unsigned higher than or same as immediate	CMP<cc> (immediate) Higher or same
	Compare unsigned higher than or same as wide elements	CMP<cc> (wide elements) Higher or same
	Compare unsigned higher than or same as vector	CMP<cc> (vectors) Higher or same
CMPLE	Compare signed less than or equal to immediate	CMP<cc> (immediate) Less than or equal
	Compare signed less than or equal to wide elements	CMP<cc> (wide elements) Less than or equal
	Compare signed less than or equal to vector	CMPLE (vectors)
CMPLO	Compare unsigned lower than immediate	CMP<cc> (immediate) Lower
	Compare unsigned lower than 64-bit wide elements	CMP<cc> (wide elements) Lower
	Compare unsigned lower than vector	CMPLO (vectors)
CMPLS	Compare unsigned lower or same as immediate	CMP<cc> (immediate) Lower or same
	Compare unsigned lower or same as wide elements	CMP<cc> (wide elements) Lower or same
	Compare unsigned lower or same as vector	CMPLS (vectors)
CMPLT	Compare signed less than immediate	CMP<cc> (immediate) Less than
	Compare signed less than wide elements	CMP<cc> (wide elements) Less than
	Compare signed less than vector	CMPLT (vectors)
CMPNE	Compare not equal to immediate	CMP<cc> (immediate) Not equal
	Compare not equal to wide elements	CMP<cc> (wide elements) Not equal
	Compare not equal to vector	CMP<cc> (vectors) Not equal

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.9.2 SVE Vector address calculation

The Vector address calculation instructions compute vectors of addresses and addresses of vectors. This includes instructions to add a multiple of the current vector length or predicate register length, in bytes, to a general-purpose register.

The `ADR` instruction is an integer arithmetic operation that is used to calculate a vector of 64-bit or 32-bit addresses.

The `ADR` destination vector elements are computed by the addition of the corresponding elements in the source vectors, with an optional sign or zero extension and optional bitwise left shift of 1-3 bits applied to the final operands. This can be considered as the addition of a vector base and a scaled vector index.

The `ADR` instruction computes a vector of 32-bit addresses by the addition of a 32-bit base and a scaled 32-bit unsigned index. The ADR instruction computes a vector of 64-bit addresses by one of:

- Addition of a 64-bit base and a scaled 64-bit unsigned index.
- Addition of a 64-bit base and a scaled, zero-extended 32-bit index.
- Addition of a 64-bit base and a scaled, sign-extended 32-bit index.

Table C3-160 Vector address calculation instructions

Mnemonic	Instruction	See
`ADDPL`	Add multiple of predicate register length, in bytes, to scalar register	ADDPL
`ADDVL`	Add multiple of vector length, in bytes, to scalar register	ADDVL
`ADR`	Calculate vector of addresses	ADR
`RDVL`	Read multiple of vector register length, in bytes, to scalar register	RDVL

C3.9.3 SVE Bitwise logical operations

The Bitwise logical operations instructions perform bitwise logical operations on vectors. Where operations are unpredicated, the operations are independent of the element size.

Table C3-161 Bitwise logical operations

Mnemonic	Instruction	See
`AND`	Bitwise AND vectors (predicated)	AND (vectors, predicated)
	Bitwise AND vectors (unpredicated)	AND (vectors, unpredicated)
	Bitwise AND with immediate	AND (immediate)
`BIC`	Bitwise clear with vector (predicated)	BIC (vectors, predicated)
	Bitwise clear with vector (unpredicated)	BIC (vectors, unpredicated)
	Bitwise clear using immediate	BIC (immediate)
`DUPM`	Broadcast bitmask immediate to vector (unpredicated)	DUPM
`EON`	Bitwise exclusive-OR with inverted immediate	EON
`EOR`	Bitwise exclusive-OR vectors (predicated)	EOR (vectors, predicated)
	Bitwise exclusive-OR vectors (unpredicated)	EOR (vectors, unpredicated)
	Bitwise exclusive-OR with immediate	EOR (immediate)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
MOV	Move bitmask immediate to vector	MOV
	Move vector register	MOV (vector, unpredicated)
NOT	Bitwise invert vector	NOT (vector)
ORN	Bitwise OR with inverted immediate	ORN (immediate)
ORR	Bitwise OR vectors (predicated)	ORR (vectors, predicated)
	Bitwise OR vectors (unpredicated)	ORR (vectors, unpredicated)
	Bitwise OR with immediate	ORR (immediate)

C3.9.4 SVE Bitwise shift, reverse, and count

Bitwise shifts, reversals, and counts within vector elements.

Shift counts saturate at the number of bits per element, rather than being used modulo the element size. If modulo behavior is required, then the modulus must be computed separately.

The wide element variants of the bitwise shift instructions allow a packed vector of narrower elements to be shifted by wider 64-bit shift amounts. These instructions treat the second source vector as having a fixed 64-bit doubleword element size and shift each narrow element of the first source vector by the corresponding vertically-aligned wide element of the second source vector. For example, if the first source vector contained 8-bit byte elements, then 8-bitelement[0] to element[7] of the first vector are shifted by 64-bit element[0] of the second source vector, 8-bit element [8] to element[15] by 64-bit element[1], and so on. All 64 bits of the wide shift amount are significant.

Table C3-162 Bitwise shift, reverse, and count instructions

Mnemonic	Instruction	See
ASR	Arithmetic shift right by immediate (predicated)	ASR (immediate, predicated)
	Arithmetic shift right by immediate (unpredicated)	ASR (immediate, unpredicated)
	Arithmetic shift right by wide elements (predicated)	ASR (wide elements, predicated)
	Arithmetic shift right by wide elements (unpredicated)	ASR (wide elements, unpredicated)
	Arithmetic shift right by vector	ASR (vectors)
ASRD	Arithmetic shift right for divide by immediate	ASRD
ASRR	Reversed arithmetic shift right by vector	ASRR
CLS	Count leading sign bits	CLS
CLZ	Count leading zero bits	CLZ
CNT	Count nonzero bits	CNT
LSL	Logical shift left by immediate (predicated)	LSL (immediate, predicated)
	Logical shift left by immediate (unpredicated)	LSL (immediate, unpredicated)
	Logical shift left by wide elements (predicated)	LSL (wide elements, predicated)
	Logical shift left by wide elements (unpredicated)	LSL (wide elements, unpredicated)
	Logical shift left by vector	LSL (vectors)
LSLR	Reversed logical shift left by vector	LSLR
LSR	Logical shift right by immediate (predicated)	LSR (immediate, predicated)
	Logical shift right by immediate (unpredicated)	LSR (immediate, unpredicated)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Logical shift right by wide elements (predicated)	LSR (wide elements, predicated)
	Logical shift right by wide elements (unpredicated)	LSR (wide elements, unpredicated)
	Logical shift right by vector	LSR (vectors)
LSRR	Reversed logical shift right by vector	LSRR
RBIT	Reverse bits	RBIT

C3.9.5 SVE Vector floating-point operations

The following instructions operate on floating-point data within a vector.

C3.9.5.1 SVE Floating-point arithmetic

The Floating-point arithmetic instructions perform arithmetic operations on vectors containing floating-point element values.

Table C3-163 Floating-point arithmetic instructions

Mnemonic	Instruction	See
FABD	Floating-point absolute difference	FABD
FABS	Floating-point absolute value	FABS
FADD	Floating-point add (immediate)	FADD (immediate)
	Floating-point add (predicated)	FADD (vectors, predicated)
	Floating-point add (unpredicated)	FADD (vectors, unpredicated)
FDIV	Floating-point divide	FDIV
FDIVR	Floating-point reversed divide	FDIVR
FMAX	Floating-point maximum with immediate	FMAX (immediate)
	Floating-point maximum vectors	FMAX (vectors)
FMAXNM	Floating-point maximum number with immediate	FMAXNM (immediate)
	Floating-point maximum number vectors	FMAXNM (vectors)
FMIN	Floating-point minimum with immediate	FMIN (immediate)
	Floating-point minimum vectors	FMIN (vectors)
FMINNM	Floating-point minimum number with immediate	FMINNM (immediate)
	Floating-point minimum number vectors	FMINNM (vectors)
FMUL	Floating-point multiply by immediate	FMUL (immediate)
	Floating-point multiply vectors (predicated)	FMUL (vectors, predicated)
	Floating-point multiply vectors (unpredicated)	FMUL (vectors, unpredicated)
FMULX	Floating-point multiply-extended	FMULX
FNEG	Floating-point negate	FNEG
FRECPE	Floating-point reciprocal estimate	FRECPE
FRECPS	Floating-point reciprocal step	FRECPS
FRECPX	Floating-point reciprocal exponent	FRECPX

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FRSQRTE	Floating-point reciprocal square root estimate	FRSQRTE
FRSQRTS	Floating-point reciprocal square root step	FRSQRTS
FSCALE	Floating-point adjust exponent by vector	FSCALE
FSQRT	Floating-point square root	FSQRT
FSUB	Floating-point subtract immediate	FSUB (immediate)
	Floating-point subtract vectors (predicated)	FSUB (vectors, predicated)
	Floating-point subtract vectors (unpredicated)	FSUB (vectors, unpredicated)
FSUBR	Floating-point reversed subtract from immediate	FSUBR (immediate)
	Floating-point reversed subtract vectors	FSUBR (vectors)

C3.9.5.2 SVE Floating-point multiply accumulate

The Floating-point multiply accumulate instructions perform floating-point fused multiply-add or multiply-subtract operations and their negated forms. There are two groups of these instructions, as follows:

- Instructions where the result of the operation is written to the addend register.
  - Supported instructions are: FMLA, FMLS, FNMLA, FNMLS.
- Instructions where the result of the operation is written to the multiplicand register.
  - Supported instructions are: FMAD, FMSB, FNMAD, FNMSB.

Table C3-164 Floating-point multiply accumulate instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add vectors, writing to the addend	FMLA (indexed) FMLA (vectors)
FMLS	Floating-point fused multiply-subtract vectors, writing to the addend	FMLS (indexed) FMLS (vectors)
FNMLA	Floating-point negated fused multiply-add vectors, writing to the addend	FNMLA
FNMLS	Floating-point negated fused multiply-subtract vectors, writing to the addend	FNMLS
FMAD	Floating-point fused multiply-add vectors, writing to the multiplicand	FMAD
FMSB	Floating-point fused multiply-subtract vectors, writing to the multiplicand	FMLSLB (indexed) FMLSLB (vectors)
FNMAD	Floating-point negated fused multiply-add vectors, writing to the multiplicand	FNMAD
FNMSB	Floating-point negated fused multiply-subtract vectors, writing to the multiplicand	FNMSB

C3.9.5.3 SVE Floating-point complex arithmetic

The Floating-point complex arithmetic instructions perform arithmetic on vectors containing floating-point complex numbers as interleaved pairs of elements, where the even-numbered elements contain the real components and the odd-numbered elements contain the imaginary components.

The FCADD instructions rotate the complex numbers in the second source vector by 90 degrees or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, before adding active pairs of elements to the corresponding elements of the first source vector in a destructive manner.

The FCMLA instructions perform a transformation of the operands to allow the creation of multiply-add or multiply-subtract operations on complex numbers by combining two of the instructions. The transformations performed are as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The complex numbers in the second source vector, considered in polar form, are rotated by 0 degrees or 180 degrees before multiplying by the duplicated real components of the first source vector.
- The complex numbers in the second source vector, considered in polar form, are rotated by 90 degrees or 270 degrees before multiplying by the duplicated imaginary components of the first source vector.

The resulting products are then added to the corresponding components of the destination and addend vector, without intermediate rounding.

Two FCMLA instructions can be used as follows:

```
FCMLA Zda.S, Pg/M, Zn.S, Zm.S, #A
...
FCMLA Zda.S, Pg/M, Zn.S, Zm.S, #B
```

For example, some meaningful combinations of A and B are:

- A=0, B=90. In this case, the two vectors of complex numbers in Zn and Zm are multiplied and the products are added to the complex numbers in Zda.
- A=0, B=270. In this case, the conjugates of the complex numbers in Zn are multiplied by the complex numbers in Zm and the products are added to the complex numbers in Zda.
- A=180, B=270. In this case, the two vectors of complex numbers in Zn and Zm are multiplied and the products are subtracted from the complex numbers in Zda.
- A=180, B=90. In this case, the conjugates of the complex numbers in Zn are multiplied by the complex numbers in Zm and the products are subtracted from the complex numbers in Zda.

Note

The lack of intermediate rounding can give unexpected results in certain cases relative to a traditional sequence of independent multiply, add, and subtract instructions.
In addition, when using these instructions, the behavior of calculations such as (∞+∞i) multiplied by (0+i) is (NaN+NaNi), rather than the result expected by ISO C, which is complex. The expectation is that these instructions are only used in situations where the effect of differences in the rounding and handling of infinities are not material to the calculation.

Table C3-165 Floating-point complex arithmetic instructions

Mnemonic	Instruction	See
FCADD	Floating-point complex add with rotate	FCADD
FCMLA	Floating-point complex multiply-add with rotate	FCMLA (vectors)

C3.9.5.4 SVE Floating-point rounding and conversion

The Floating-point rounding and conversion instructions change floating-point size and precision, round floating-point to integral floating-point with explicit rounding mode, and convert floating-point to or from integer format.

Table C3-166 Floating-point rounding and conversion instructions

Mnemonic	Instruction	See
BFCVT	Floating-point down convert to BFloat16 format	BFCVT
BFCVTNT	Floating-point down convert and narrow to BFloat16 format (top, predicated)	BFCVTNT

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FCVT	Floating-point convert precision	FCVT
FCVTZS	Floating-point convert to signed integer, rounding toward zero	FCVTZS
FCVTZU	Floating-point convert to unsigned integer, rounding toward zero	FCVTZU
FRINTA	Floating-point round to integral value, to nearest with ties away from zero	FRINT<r> Nearest with ties to away
FRINTI	Floating-point round to integral value, using the current rounding mode	FRINT<r> Current mode
FRINTM	Floating-point round to integral value, toward minus infinity	FRINT<r> Toward minus infinity
FRINTN	Floating-point round to integral value, to nearest with ties to even	FRINT<r> Nearest with ties to even
FRINTP	Floating-point round to integral value, toward plus infinity	FRINT<r> Toward plus infinity
FRINTX	Floating-point round to integral value exact, using the current rounding mode	FRINT<r> Current mode signalling inexact
FRINTZ	Floating-point round to integral value, toward zero	FRINT<r> Toward zero
SCVTF	Signed integer convert to floating-point	SCVTF
UCVTF	Unsigned integer convert to floating-point	UCVTF

C3.9.5.5 SVE Floating-point comparisons

The Floating-point comparison instructions compare active floating-point element values in the first source vector with corresponding elements in the second vector or with the immediate value +0.0. The Boolean result of each comparison is placed in the corresponding element of the destination predicate. *Inactive elements* in the destination predicate register are set to FALSE. Floating-point vector comparisons do not set the condition flags.

Table C3-167 Floating-point comparison instructions

Mnemonic	Instruction	See
FACGE	Floating-point absolute compare greater than or equal	FAC<cc> - Greater than or equal
FACGT	Floating-point absolute compare greater than	FAC<cc> - Greater than
FACLE	Floating-point absolute compare less than or equal	FACLE
FACLT	Floating-point absolute compare less than	FACLT
FCMEQ	Floating-point compare equal to zero	FCM<cc> (zero) - Equal
	Floating-point compare equal to vector	FCM<cc> (vectors) - Equal
FCMGE	Floating-point compare greater than or equal to zero	FCM<cc> (zero) - Greater than or equal
	Floating-point compare greater than or equal to vector	FCM<cc> (vectors) - Greater than or equal
FCMGT	Floating-point compare greater than zero	FCM<cc> (zero) - Greater than
	Floating-point compare greater than vector	FCM<cc> (vectors) - Greater than
FCMLE	Floating-point compare less than or equal to zero	FCM<cc> (zero) - Less than or equal
	Floating-point compare less than or equal to vector	FCMLE (vectors)
FCMLT	Floating-point compare less than zero	FCM<cc> (zero) - Less than
	Floating-point compare less than vector	FCMLT (vectors)
FCMNE	Floating-point compare not equal to zero	FCM<cc> (zero) - Not equal
	Floating-point compare not equal to vector	FCM<cc> (vectors) - Not equal
FCMUO	Floating-point unordered vectors	FCM<cc> (vectors) - Unordered

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See

C3.9.5.6 SVE Floating-point transcendental acceleration

The Floating-point transcendental instructions accelerate calculations of sine, cosine, and exponential functions for vectors containing floating-point element values.

The trigonometric instructions accelerate the calculation of a polynomial series approximation for the sine and cosine functions. The exponential instruction accelerates the polynomial series calculation of the exponential function.

Table C3-168 Floating-point transcendental instructions

Mnemonic	Instruction	See
`FTMAD`	Floating-point trigonometric multiply-add coefficient	FTMAD
`FTSMUL`	Floating-point trigonometric starting value	FTSMUL
`FTSSEL`	Floating-point trigonometric select coefficient	FTSSEL
`FEXPA`	Floating-point exponential accelerator	FEXPA

C3.9.5.7 SVE Floating-point indexed multiples

The Floating-point indexed multiples instructions multiply all floating-point elements within each 128-bit segment of the first source vector by the single numbered element within the corresponding segment of the second source vector.

For the `FMLA` and `FMLS` instructions, the products are destructively added or subtracted from the corresponding elements of the addend and destination vector, without intermediate rounding.

Table C3-169 Floating-point indexed multiples instructions

Mnemonic	Instruction	See
`FMLA`	Floating-point fused multiply-add by indexed elements	FMLA (indexed)
`FMLS`	Floating-point fused multiply-subtract by indexed elements	FMLS (indexed)
`FMUL`	Floating-point multiply by indexed elements	FMUL (indexed)

C3.9.5.8 SVE Floating-point matrix multiply-accumulate and companion instructions

The floating-point matrix multiply-accumulate and companion instructions facilitate matrix multiplication computation.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. The segments are organized as follows:

- In the first source vector, a 2x2 matrix in row-by-row order.
- In the second source vector, a 2x2 matrix in column-by-column order.
- In the destination vector, a 2x2 matrix in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment. For the double-precision matrix multiply-accumulate instructions, the vector segment length and minimum vector length is 256 bits. Double-precision matrix multiply-accumulate instructions are not supported when the vector length is 128 bits. For the single-precision matrix multiply-accumulate instruction, the vector segment length is 128 bits.

The floating-point matrix multiply-accumulate instructions strictly define the order of accumulations, and the multiplications and additions are not fused, so intermediate rounding is performed after every multiplication and every addition.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The companion instructions support data rearrangements in vector registers as required by the double-precision matrix multiply-accumulate instructions.

The following table shows the floating-point matrix multiplication instructions and companion instructions that are implemented by FEAT_F64MM:

Table C3-170 Floating-point matrix multiply instructions

Mnemonic	Instruction	See
FMMLA	Floating-point matrix multiply-accumulate into 2x2 matrix (double-precision)	FMMLA
LD1ROB	Contiguous load and replicate thirty-two bytes, scalar plus scalar	LD1ROB (scalar plus scalar)
	Contiguous load and replicate thirty-two bytes, scalar plus immediate	LD1ROB (scalar plus immediate)
LD1ROD	Contiguous load and replicate four doublewords, scalar plus scalar	LD1ROD (scalar plus scalar)
	Contiguous load and replicate four doublewords, scalar plus immediate	LD1ROD (scalar plus immediate)
LD1ROH	Contiguous load and replicate sixteen halfwords, scalar plus scalar	LD1ROH (scalar plus scalar)
	Contiguous load and replicate sixteen halfwords, scalar plus immediate	LD1ROH (scalar plus immediate)
LD1ROW	Contiguous load and replicate eight words, scalar plus scalar	LD1ROW (scalar plus scalar)
	Contiguous load and replicate eight words, scalar plus immediate	LD1ROW (scalar plus immediate)
TRN1, TRN2	Interleave even or odd 128-bit elements from two vectors	TRN1, TRN2 (vectors)
UZP1, UZP2	Concatenate even or odd 128-bit elements from two vectors	UZP1, UZP2 (vectors)
ZIP1, ZIP2	Interleave 128-bit elements from two half vectors	ZIP1, ZIP2 (vectors)

The following table shows the floating-point matrix multiplication instruction, which is implemented by FEAT_F32MM:

Mnemonic	Instruction	See
FMMLA	Floating-point matrix multiply-accumulate into 2x2 matrix (single-precision)	FMMLA

C3.9.5.9 SVE BFloat16 floating-point multiply-add

The BFloat16 floating-point multiply-add instructions perform an implicit conversion of the bottom (even-numbered) or top (odd-numbered) BFloat16 source elements to IEEE 754 single-precision floating-point format before performing a fused multiply-add without intermediate rounding to the overlapping single-precision destination element. These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

Table C3-172 SVE BFloat16 floating-point multiply-add instructions

Mnemonic	Instruction	See
BFMLALB	BFloat16 floating-point widening multiply accumulate long bottom (vector and indexed forms)	BFMLALB (vectors) BFMLALB (indexed)
BFMLALT	BFloat16 floating-point widening multiply accumulate long top (vector and indexed forms)	BFMLALT (vectors) BFMLALT (indexed)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
BFMLSLB	BFloat16 floating-point multiply-subtract long from single-precision (bottom, indexed)	BFMLSLB (indexed)
	BFloat16 floating-point multiply-subtract long from single-precision (bottom)	BFMLSLB (vectors)
BFMLSLT	BFloat16 floating-point multiply-subtract long from single-precision (top, indexed)	BFMLSLT (indexed)
	BFloat16 floating-point multiply-subtract long from single-precision (top)	BFMLSLT (vectors)

C3.9.5.10 SVE BFloat16 floating-point dot product

The BFloat16 floating-point dot product instruction, BFDOT, performs an implicit conversion of vectors of BFloat16 input values to IEEE 754 single-precision floating-point format.

The BFloat16 dot product instructions delimit their source vectors into pairs of BFloat16 elements.

Table C3-173 SVE BFloat16 floating-point dot product instructions

Mnemonic	Instruction	See
BFDOT	BFloat16 floating-point dot product (vector and indexed forms)	BFDOT (vectors) BFDOT (indexed)

C3.9.5.11 SVE BFloat16 floating-point matrix multiply-accumulate

The BFloat16 floating-point matrix multiply-accumulate instruction, BFMMLA, is implemented by FEAT_BF16.

This instruction delimits the source and destination vectors into 128-bit segments. The 128-bit segments are organized as follows:

- In the first source vector, a 2x4 matrix of BFloat16 values in row-by-row order.
- In the second source vector, a 4x2 matrix of BFloat16 values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per source vector segment and accumulated into the corresponding destination vector segment. This corresponds to accumulating two 2-way BFloat16 widening dot products into each single-precision destination element, following the numeric behaviors described for BFDOT instruction in SVE BFloat16 floating-point dot product.

Table C3-174 SVE BFloat16 floating-point matrix multiply instruction

Mnemonic	Instruction	See
BFMMLA	BFloat16 floating-point matrix multiply-accumulate	BFMMLA

C3.9.5.12 SVE BFloat16 floating-point convert

The BFloat16 floating-point convert instructions perform accurately rounded down-conversion of IEEE 754 single-precision source vector elements to BFloat16 format.

The BFCVT instruction places its BFloat16 results in the bottom or even-numbered 16-bit elements of the destination vector, and sets the top or odd-numbered elements to zero.

The BFCVTNT instruction places its BFloat16 results in the top or odd-numbered 16-bit elements of the destination vector, leaving the bottom or even-numbered elements unchanged.

These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-175 SVE BFloat16 floating-point convert instructions

Mnemonic	Instruction	See
BFCVT	Floating-point down convert to BFloat16 format (predicated)	BFCVT
BFCVTNT	Floating-point down convert and narrow to BFloat16 (top, predicated)	BFCVTNT

C3.9.6 Predicate operations

The Predicate instructions perform operations that manipulate the predicate registers.

Some of these instructions are insensitive to the predicate element size and specify an explicit byte element size qualifier, .B, but an assembler must accept any qualifier, or none.

C3.9.6.1 SVE Predicate initialization

The Predicate initialization instructions initialize predicate elements.Predicate elements can be initialized to be FALSE, or to be TRUE when their element number is less than:

- A fixed number of elements from the following range: VL1-VL8, VL16, VL32, VL64, VL128 or VL256.
- The largest power of two elements, POW2.
- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, which is implicitly a multiple of two.

Unspecified or out of range constraint encodings generate a predicate with values that are all FALSE and do not cause an Undefined Instruction exception.

Table C3-176 Predicate initialization instructions

Mnemonic	Instruction	See
PFALSE	Set all predicate elements to FALSE	PFALSE
PTRUE	Initialize predicate elements from named constraint	PTRUE (predicate)
PTRUES	Initialize predicate elements from named constraint, setting the condition flags	PTRUES

C3.9.6.2 SVE Predicate move operations

The Predicate move instructions operate on all bits of the predicate registers, implying a fixed, 1-bit predicate element size. The flag-setting variants set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the flag-setting variants should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

Table C3-177 Predicate move instructions

Mnemonic	Instruction	See
SEL	Select predicate elements from two predicates	SEL (predicates)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
MOV	Move predicate elements (predicated, merging)	MOV (predicate, predicated, merging)
	Move predicate elements (predicated, zeroing)	MOV (predicate, predicated, zeroing)
	Move predicate elements (unpredicated)	MOV
MOVS	Move predicate elements, setting the condition flags (predicated)	MOVS (predicated)
	Move predicate elements, setting the condition flags (unpredicated)	MOVS (unpredicated)

C3.9.6.3 SVE Predicate logical operations

The Predicate logical operation instructions perform bitwise logical operations on predicate registers that operate on all bits of the register, implying a fixed, 1-bit predicate element size. The flag-setting variants set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero. Inactive elements in the destination Predicate register are set to zero, except for PTEST which does not specify a destination register. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the flag-setting variants should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

Table C3-178 Predicate logical operation instructions

Mnemonic	Instruction	See
AND	Bitwise AND predicates	AND (predicates)
ANDS	Bitwise AND predicates, setting the condition flags	ANDS
BIC	Bitwise clear predicates	BIC (predicates)
BICS	Bitwise clear predicates, setting the condition flags	BICS
EOR	Bitwise exclusive-OR predicates	EOR (predicates)
EORS	Bitwise exclusive-OR predicates, setting the condition flags	EORS
NAND	Bitwise NAND predicates	NAND
NANDS	Bitwise NAND predicates, setting the condition flags	NANDS
NOR	Bitwise NOR predicates	NOR
NORS	Bitwise NOR predicates, setting the condition flags	NORS
NOT	Bitwise invert predicate	NOT (predicate)
NOTS	Bitwise invert predicate, setting the condition flags	NOTS
ORN	Bitwise OR inverted predicate	ORN (predicates)
ORNS	Bitwise OR inverted predicate, setting the condition flags	ORNS
ORR	Bitwise OR predicates	ORR (predicates)
ORRS	Bitwise OR predicates, setting the condition flags	ORRS
PTEST	Test predicate value, setting the condition flags	PTEST

C3.9.6.4 FFR predicate handling

The FFR predicate handling instructions work with SVE First-fault and Non-fault loads using the FFR to determine which elements have been successfully loaded and which remain to be loaded on a subsequent iteration.

The RDFFRS instruction sets the N, Z, and C condition flags based on the predicate result, and sets the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the RDFFRS instruction should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-179 FFR predicate handling instructions

Mnemonic	Instruction	See
RDFFR	Return predicate of successfully loaded elements (unpredicated)	RDFFR (unpredicated)
	Return predicate of successfully loaded elements (predicated)	RDFFR (predicated)
RDFFRS	Return predicate of successfully loaded elements, setting the condition flags (predicated)	RDFFRS
SETFFR	Initialize the First-fault register to all TRUE	SETFFR
WRFFR	Write a predicate register to the First-fault register	WRFFR

C3.9.6.5 Predicate count

The Predicate count instructions count either the number of Active predicate elements that are set to TRUE, or the number of elements implied by a named predicate constraint. The count can be placed in a general-purpose register, or used to increment or decrement a vector or general-purpose register.

Signed or unsigned saturating variants handle cases where, for example, an increment might cause a vectorized scalar loop index to overflow and therefore never satisfy a loop termination condition that compares it with a limit that is close to the maximum integer value.

The named predicate constraint limits the number of elements to:

- A fixed number of elements from the following range: VL1-VL8, VL16, VL32, VL64, VL128 or VL256.
- The largest power of two elements, POW2.
- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, implicitly a multiple of two.

Unspecified or out of range predicate constraint encodings generate a zero element count and do not cause an Undefined Instruction exception.

Table C3-180 Predicate count instructions

Mnemonic	Instruction	See
CNTB	Set scalar to multiple of 8-bit predicate constraint element count	CNTB, CNTD, CNTH, CNTW
CNTH	Set scalar to multiple of 16-bit predicate constraint element count	CNTB, CNTD, CNTH, CNTW
CNTW	Set scalar to multiple of 32-bit predicate constraint element count	CNTB, CNTD, CNTH, CNTW
CNTD	Set scalar to multiple of 64-bit predicate constraint element count	CNTB, CNTD, CNTH, CNTW
CNTP	Set scalar to the number of Active predicate elements that are TRUE	CNTP (predicate)
DECB	Decrement scalar by multiple of 8-bit predicate constraint element count	DECB, DECD, DECH, DECW (scalar)
DECH	Decrement scalar by multiple of 16-bit predicate constraint element count	DECB, DECD, DECH, DECW (scalar)
	Decrement vector by multiple of 16-bit predicate constraint element count	DECD, DECH, DECW (vector)
DECW	Decrement scalar by multiple of 32-bit predicate constraint element count	DECB, DECD, DECH, DECW (scalar)
	Decrement vector by multiple of 32-bit predicate constraint element count	DECD, DECH, DECW (vector)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
DECD	Decrement scalar by multiple of 64-bit predicate constraint element count	DECB, DECD, DECH, DECW (scalar)
	Decrement vector by multiple of 64-bit predicate constraint element count	DECD, DECH, DECW (vector)
DECP	Decrement scalar by the number of predicate elements that are TRUE	DECP (scalar)
	Decrement vector by the number of Active predicate elements that are TRUE	DECP (vector)
INCB	Increment scalar by multiple of 8-bit predicate constraint element count	INCB, INCD, INCH, INCW (scalar)
INCH	Increment scalar by multiple of 16-bit predicate constraint element count	INCB, INCD, INCH, INCW (scalar)
	Increment vector by multiple of 16-bit predicate constraint element count	INCD, INCH, INCW (vector)
INCW	Increment scalar by multiple of 32-bit predicate constraint element count	INCB, INCD, INCH, INCW (scalar)
	Increment vector by multiple of 32-bit predicate constraint element count	INCD, INCH, INCW (vector)
INCD	Increment scalar by multiple of 64-bit predicate constraint element count	INCB, INCD, INCH, INCW (scalar)
	Increment vector by multiple of 64-bit predicate constraint element count	INCD, INCH, INCW (vector)
INCP	Increment scalar by the number of predicate elements that are TRUE	INCP (scalar)
	Increment vector by the number of predicate elements that are TRUE	INCP (vector)
SQDECB	Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count	SQDECB
SQDECH	Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count	SQDECH (scalar)
	Signed saturating decrement vector by multiple of 16-bit predicate constraint element count	SQDECH (vector)
SQDECW	Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count	SQDECW (scalar)
	Signed saturating decrement vector by multiple of 32-bit predicate constraint element count	SQDECW (vector)
SQDECD	Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count	SQDECD (scalar)
	Signed saturating decrement vector by multiple of 64-bit predicate constraint element count	SQDECD (vector)
SQDECP	Signed saturating decrement scalar the number of predicate elements that are TRUE	SQDECP (scalar)
	Signed saturating decrement vector by the number of predicate elements that are TRUE	SQDECP (vector)
SQINCB	Signed saturating increment scalar by multiple of 8-bit predicate constraint element count	SQINCB
SQINCH	Signed saturating increment scalar by multiple of 16-bit predicate constraint element count	SQINCH (scalar)
	Signed saturating increment vector by multiple of 16-bit predicate constraint element count	SQINCH (vector)
SQINCW	Signed saturating increment scalar by multiple of 32-bit predicate constraint element count	SQINCW (scalar)
	Signed saturating increment vector by multiple of 32-bit predicate constraint element count	SQINCW (vector)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SQINCD	Signed saturating increment scalar by multiple of 64-bit predicate constraint element count	SQINCD (scalar)
	Signed saturating increment vector by multiple of 64-bit predicate constraint element count	SQINCD (vector)
SQINCP	Signed saturating increment scalar by the number of predicate elements that are TRUE	SQINCP (scalar)
	Signed saturating increment vector by the number of predicate elements that are TRUE	SQINCP (vector)
UQDECB	Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count	UQDECB
UQDECH	Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count	UQDECH (scalar)
	Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count	UQDECH (vector)
UQDECW	Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count	UQDECW (scalar)
	Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count	UQDECW (vector)
UQDECD	Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count	UQDECD (scalar)
	Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count	UQDECD (vector)
UQDECP	Unsigned saturating decrement scalar by the number of predicate elements that are TRUE	UQDECP (scalar)
	Unsigned saturating decrement vector by the number of predicate elements that are TRUE	UQDECP (vector)
UQINCB	Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count	UQINCB
UQINCH	Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count	UQINCH (scalar)
	Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count	UQINCH (vector)
UQINCW	Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count	UQINCW (scalar)
	Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count	UQINCW (vector)
UQINCD	Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count	UQINCD (scalar)
	Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count	UQINCD (vector)
UQINCP	Unsigned saturating increment scalar by the number of predicate elements that are TRUE	UQINCP (scalar)
	Unsigned saturating increment vector by the number of predicate elements that are TRUE	UQINCP (vector)

C3.9.7 Loop control

These instructions control counted vector loops and vector loops with data-dependent termination conditions. These instructions create a loop partition predicate with *Active elements* set to TRUE up to the point where the loop should

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

terminate, and FALSE thereafter. Two loop concepts are supported, simple loops and data-dependent loops.

C3.9.7.1 Simple loops

An up-counting `WHILE` instruction that increments the value of the first scalar operand and compares the value with a second, fixed scalar operand. The instruction generates a destination predicate with all of the following characteristics:

- The predicate elements starting from the lowest numbered element are true while the comparison is true.
- The predicate elements thereafter, up to the highest numbered element, are false when the comparison becomes false.

All 32 bits or 64 bits of the scalar operands are significant for the purposes of comparison. The full 32-bit or 64-bit value of the first operand is incremented by 1 for each destination predicate element, irrespective of the element size. The first general-purpose register operand is not updated.

If all of the following occur, a comparison can never fail, resulting in an all-true predicate:

- The comparison includes an equality test.
- The second scalar operand is equal to the maximum integer value of the selected size and type of comparison.

The N, Z, C, and V condition flags are unconditionally set to control a subsequent conditional branch.

Table C3-181 Single loop instructions

Mnemonic	Instruction	See
`WHILELE`	While incrementing signed scalar less than or equal to scalar	WHILELE (predicate)
`WHILELO`	While incrementing unsigned scalar lower than scalar	WHILELO (predicate)
`WHILELS`	While incrementing unsigned scalar lower than or the same as scalar	WHILELS (predicate)
`WHILELT`	While incrementing signed scalar less than scalar	WHILELT (predicate)

C3.9.7.2 Data-independent loops

For data-dependent termination conditions, it is necessary to convert the result of a vector comparison into a loop partition predicate. The new partition truncates the current vector partition immediately before or after the first active TRUE comparison. The N, Z, C, and V condition flags are optionally set to control a subsequent conditional branch.

The `BRKA` instructions set active destination predicate elements to TRUE up to and including the first active TRUE element in their source predicate register, setting subsequent elements to FALSE.

The `BRKB` instructions set active destination predicate elements to TRUE up to but excluding the first active TRUE element in their source predicate register, setting subsequent elements to FALSE.

The `BRKPA` and `BRKPB` instructions propagate the result of a previous `BRKB` or `BRKPB` instruction, by setting their destination predicate register to all FALSE if the *Last active element* of their first source predicate register is not TRUE, but otherwise generate the destination predicate from their second source predicate as described for the `BRKA` and `BRKB` instructions.

The BRKN instructions propagate the result of a previous `BRKB` or `BRKPB` instruction by setting the destination predicate register to all FALSE if the *Last active element* of their first source predicate register is not TRUE, but otherwise leave the destination predicate unchanged. The destination and second source predicate must have been created by another instruction, such as `RDFFR` or `WHILE`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-182 Data-independent loop instructions

Mnemonic	Instruction	See
BRKA	Break after the first true condition	BRKA
BRKAS	Break after the first true condition, setting the condition flags	BRKAS
BRKB	Break before the first true condition	BRKB
BRKBS	Break before the first true condition, setting the condition flags	BRKBS
BRKN	Propagate break to next partition	BRKN
BRKNS	Propagate break to next partition, setting the condition flags	BRKNS
BRKPA	Break after the first true condition, propagating from previous partition	BRKPA
BRKPAS	Break after the first true condition, propagating from previous partition, setting the condition flags	BRKPAS
BRKPB	Break before the first true condition, propagating from the previous partition	BRKPB
BRKPBS	Break before the first true condition, propagating from the previous partition, setting the condition flags	BRKPBS

C3.9.7.3 Serialized operations

The Serialized operation instructions permit *Active elements* within a vector to be processed sequentially without unpacking the vector. The condition flags are unconditionally set to control a subsequent conditional branch.

Table C3-183 Serialized operation instructions

Mnemonic	Instruction	See
PFIRST	Set the *First active element* to TRUE	PFIRST
PNEXT	Find next *Active element*	PNEXT
CTERMEQ	Compare and terminate loop when equal	CTERMEQ, CTERMNE
CTERMNE	Compare and terminate loop when not equal	CTERMEQ, CTERMNE

C3.9.8 SVE Move operations

C3.9.8.1 Element move and broadcast

The Element move and broadcast instructions copy data from scalar registers, immediate values, and other vectors to all vector elements or only to Active vector elements. The copied data might be in an integer or floating-point format.

Table C3-184 Element move and broadcast instructions

Mnemonic	Instruction	See
CPY	Copy signed integer immediate to Active vector elements	CPY (immediate, merging) CPY (immediate, zeroing)
	Copy general-purpose register to Active vector elements	CPY (scalar)
	Copy SIMD&FP scalar register to Active vector elements	CPY (SIMD&FP scalar)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
DUP	Broadcast signed immediate to all vector elements	DUP (immediate)
	Broadcast general-purpose register to all vector elements	DUP (scalar)
FCPY	Copy floating-point immediate to Active vector elements	FCPY
FDUP	Broadcast floating-point immediate to all vector elements	FDUP
FMOV	Move floating-point +0.0 to vector elements (unpredicated)	FMOV (zero, unpredicated)
	Move floating-point +0.0 to vector elements (predicated)	FMOV (zero, predicated)
	Move floating-point immediate to vector elements (unpredicated)	FMOV (immediate, unpredicated)
	Move floating-point immediate to vector element (predicated)	FMOV (immediate, predicated)
MOV	Move signed integer immediate to vector elements (unpredicated)	MOV (immediate, unpredicated)
	Move signed integer immediate to vector elements (predicated)	MOV (immediate, predicated, merging) MOV (immediate, predicated, zeroing)
	Move general-purpose register to vector elements (unpredicated)	MOV (scalar, unpredicated)
	Move general-purpose register to vector elements (predicated)	MOV (scalar, predicated)
	Move SIMD&FP scalar register to vector elements (unpredicated)	MOV (SIMD&FP scalar, unpredicated)
MOV	Move SIMD&FP scalar register to vector elements (predicated)	MOV (SIMD&FP scalar, predicated)
	Move vector register (unpredicated)	MOV (vector, unpredicated)
	Move vector register (predicated)	MOV (vector, predicated)
SEL	Select vector elements from two vectors	SEL (predicates) SEL (vectors)

C3.9.8.2 Element permute and shuffle

The Element permute and shuffle instructions move data between different vector elements, or between vector elements and scalar registers.

These instructions perform the following operations:

- Conditionally extract the *Last active element* of a vector or the following element.
  - The supported instructions are CLASTA and CLASTB.
- Unconditionally extract the *Last active element* of a vector or the following element.
  - The supported instructions are LASTA and LASTB.
- Variable permute instructions where the permutation is determined by the values in a predicate register or a table of element index values.
  - The supported instructions are COMPACT, SPLICE, and TBL.
- Fixed permute instructions where the form of the permutation is encoded in the instruction.
  - The supported instructions are: DUP, EXT, INSR, REV, REVB, REVH, REVW, SUNPKHI, SUNPKLO, TRN1, TRN2, UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, and ZIP2.

Table C3-185 Element permute and shuffle instructions

Mnemonic	Instruction	See
CLASTA	Conditionally extract element after the *Last active element* to general-purpose register	CLASTA (scalar)
	Conditionally extract element after the *Last active element* to SIMD&FP scalar	CLASTA (SIMD&FP scalar)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Conditionally extract element after the *Last active element* to vector	CLASTA (vectors)
CLASTB	Conditionally extract *Last active element* to general-purpose register	CLASTB (scalar)
	Conditionally extract *Last active element* to SIMD&FP scalar	CLASTB (SIMD&FP scalar)
	Conditionally extract *Last active element* to vector	CLASTB (vectors)
LASTA	Extract element after the *Last active element* to general-purpose register	LASTA (scalar)
	Extract element after the *Last active element* to SIMD&FP scalar	LASTA (SIMD&FP scalar)
LASTB	Extract *Last active element* to general-purpose register	LASTB (scalar)
	Extract *Last active element* to SIMD&FP scalar	LASTA (SIMD&FP scalar)
COMPACT	Copy Active vector elements to lower-numbered elements	COMPACT
SPLICE	Splice two vectors under predicate control	SPLICE
TBL	Programmable table lookup using vector of element indexes	TBL
DUP	Broadcast indexed vector element	DUP (indexed)
EXT	Extract vector from pair of vectors	EXT
INSR	Insert general-purpose register into shifted vector	INSR (scalar)
	Insert SIMD&FP scalar register into shifted vector	INSR (SIMD&FP scalar)
MOV	Move indexed element or SIMD&FP scalar to vector (unpredicated)	MOV (SIMD&FP scalar, unpredicated)
	Move SIMD&FP scalar register to vector elements (predicated)	MOV (SIMD&FP scalar, predicated)
REV	Reverse all elements in vector	REV (vector)
REVB	Reverse 8-bit bytes in elements	REVB, REVH, REVW
REVH	Reverse 16-bit halfwords in elements	REVB, REVH, REVW
REVW	Reverse 32-bit words in elements	REVB, REVH, REVW
TRN1	Interleave even elements from two vectors	TRN1, TRN2 (vectors)
TRN2	Interleave odd elements from two vectors	TRN1, TRN2 (vectors)
UZP1	Concatenate even elements from two vectors	UZP1, UZP2 (vectors)
UZP2	Concatenate odd elements from two vectors	UZP1, UZP2 (vectors)
ZIP1	Interleave elements from low halves of two vectors	ZIP1, ZIP2 (vectors)
ZIP2	Interleave elements from high halves of two vectors	ZIP1, ZIP2 (vectors)

C3.9.8.3 Unpacking instructions

The Unpacking instructions unpack half of the elements from the source vector register or predicate register, widen the unpacked elements to twice the width, and place the result in the destination register.

Table C3-186 Unpacking instructions

Mnemonic	Instruction	See
SUNPKHI	Unpack and sign-extend elements from high half of vector	SUNPKHI, SUNPKLO
SUNPKLO	Unpack and sign-extend elements from low half of vector	SUNPKHI, SUNPKLO
UUNPKHI	Unpack and zero-extend elements from high half of vector	UUNPKHI, UUNPKLO
UUNPKLO	Unpack and zero-extend elements from low half of vector	UUNPKHI, UUNPKLO

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
PUNPKHI	Unpack and widen elements from high half of predicate	PUNPKHI, PUNPKLO
PUNPKLO	Unpack and widen elements from low half of predicate	PUNPKHI, PUNPKLO

C3.9.8.4 Predicate permute

The Predicate instructions are used to move and permute predicate elements. These instructions generally mirror the fixed vector permutes to allow predicates to follow their data. The permutes move all of the bits in a predicate element, not just the canonical bits.

Table C3-187 Predicate instructions

Mnemonic	Instruction	See
REV	Reverse all elements in predicate	REV (predicate)
TRN1	Interleave even elements from two predicates	TRN1, TRN2 (predicates)
TRN2	Interleave odd elements from two predicates	TRN1, TRN2 (predicates)
UZP1	Select even elements from two predicates	UZP1, UZP2 (predicates)
UZP2	Select odd elements from two predicates	UZP1, UZP2 (predicates)
ZIP1	Interleave elements from low halves of two predicates	ZIP1, ZIP2 (predicates)
ZIP2	Interleave elements from high halves of two predicates	ZIP1, ZIP2 (predicates)

C3.9.9 Index vector generation

The INDEX instruction initializes a vector horizontally by setting its first element to an integer value, and then repeatedly incrementing it by a second integer value to generate the subsequent elements. Each integer value can be specified as a signed immediate or a general-purpose register.

Table C3-188 Index instructions

Mnemonic	Instruction	See
INDEX	Create index vector starting from and incremented by immediates	INDEX (immediates)
	Create index vector starting from immediate and incremented by general-purpose register	INDEX (scalars)
	Create index vector starting from general-purpose register and incremented by immediate	INDEX (scalar, immediate)
	Create index vector starting from immediate and incremented by general-purpose registers	INDEX (immediate, scalar)

C3.9.10 Move prefix

The MOVPRFX (predicated) instruction is a predicated vector move that can be combined with a predicated destructive instruction that immediately follows it, in program order, to create a single constructive operation, or to convert an instruction with merging predication to use zeroing predication.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The MOVPRFX (unpredicated) instruction is an unpredicated vector move that can be combined with a predicated or unpredicated destructive instruction that immediately follows it, in program order, to create a single constructive operation.

The *Operational information* section of an SVE instruction description indicates whether or not an instruction can be predictably prefixed by a MOVPRFX instruction. If the *Operational information* of an *SVE* instruction description does not mention MOVPRFX or if the section does not exist, then the instruction cannot be predictably prefixed by a MOVPRFX instruction.

The prefixed instruction that immediately follows a MOVPRFX instruction in program order must be an implemented SVE instruction that can be predictably prefixed by a MOVPRFX instruction, or an A64 HLT instruction, or an A64 BRK instruction. For an SVE instruction that can be predictably prefixed by a MOVPRFX instruction, all of the following apply:

- The destination register field implicitly specifies one of the source operands, which means that it is a destructive binary or ternary vector operation or unary operation with merging predication, excluding MOVPRFX.
- The destination register is the same as the MOVPRFX destination register.
- The prefixed instruction does not use the MOVPRFX destination register in any of its other source register fields, even if it has a different name but refers to the same architectural register state. For example, Z1, V1, and D1 all refer to the same architectural register.
- If the MOVPRFX instruction is predicated, then the prefixed instruction is predicated using the same *Governing predicate* register, and the maximum encoded element size is the same as the MOVPRFX element size, excluding the fixed-size 64-bit elements of the wide elements form of bitwise shift and integer compare operations.
- If the MOVPRFX instruction is unpredicated, then the prefixed instruction can use any *Governing predicate* register and element size, or it can be unpredicated. A predicated MOVPRFX cannot be used with an unpredicated instruction.

If the instruction that follows a MOVPRFX instruction is not an implemented SVE instruction that can be predictably prefixed by a MOVPRFX instruction, the two instructions behave in one of the following CONSTRAINED UNPREDICTABLE ways:

- Either or both instructions can execute with their individually described effects.
- Either instruction can generate an Undefined Instruction exception.
- Either or both instructions can execute as a NOP.
- The second instruction can execute with an UNKNOWN value for any of its source registers.
- Any register that is written by either or both instructions can be set to an unknown value.
- A control flow instruction that writes the PC can set the PC to an unknown value.

Table C3-189 Move prefix instructions

Mnemonic	Instruction	See
MOVPRFX	Move prefix (predicated)	MOVPRFX (predicated)
	Move prefix (unpredicated)	MOVPRFX (unpredicated)

Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that, for performance reasons, software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

When a MOVPRFX instruction is executed, except for PMU events SVE_MOVPRFX_SPEC, SVE_MOVPRFX_Z_SPEC, SVE_MOVPRFX_M_SPEC, and SVE_MOVPRFX_U_SPEC, 0x807C-0x807F, it is IMPLEMENTATION DEFINED for each execution of the instruction whether or not any Performance Monitor counts the instruction. This can vary dynamically for each execution of the same instruction.

When a microarchitectural operation is executed because of a MOVPRFX instruction, except for PMU events SVE_MOVPRFX_SPEC, SVE_MOVPRFX_Z_SPEC, SVE_MOVPRFX_M_SPEC, and SVE_MOVPRFX_U_SPEC, 0x807C-0x807F, it is IMPLEMENTATION DEFINED for each execution of the operation whether or not the Performance Monitor counts the operation. This can vary dynamically for each execution of the same instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.9.10.1 MOVPRFX instruction behavior in self-hosted debug

A MOVPRFX instruction can legally prefix a BRK or HLT instruction.

If a hardware breakpoint is programmed with the address of a legal MOVPRFX instruction, when any of the following events occur, the hardware breakpoint generates a Breakpoint exception:

- The MOVPRFX instruction is committed for execution.
- The combined MOVPRFX and Prefixed instruction is committed for execution.

If a hardware breakpoint is programmed with the address of an illegal MOVPRFX instruction or a Prefixed instruction, when any of the MOVPRFX instruction and Prefixed instruction are committed for execution, it is CONSTRAINED UNPREDICTABLE whether or not the hardware breakpoint generates a Breakpoint exception.

If a single-step is performed for a MOVPRFX instruction, it is CONSTRAINED UNPREDICTABLE whether the PE steps over the pair of instructions or steps over only the MOVPRFX instruction.

C3.9.11 Reduction operations

C3.9.11.1 Horizontal reductions

The Horizontal reduction instructions perform arithmetic horizontally across *Active elements* of a single source vector and deliver a scalar result.

The floating-point horizontal accumulating sum instruction, FADDA, operates strictly in order of increasing element number across a vector, using the scalar destination register as a source for the initial value of the accumulator. This preserves the original program evaluation order where non-associativity is required.

The other floating-point reductions calculate their result using a recursive pair-wise algorithm that does not preserve the original program order, but permits increased parallelism for code that does not require strict order of evaluation.

Integer reductions are fully associative, and the order of evaluation is not specified by the architecture.

Table C3-190 Horizontal reduction instructions

Mnemonic	Instruction	See
ANDV	Bitwise AND reduction, treating *Inactive elements* as all ones	ANDV
EORV	Bitwise EOR reduction, treating *Inactive elements* as zero	EORV
FADDA	Floating-point add strictly-ordered reduction, accumulating in scalar, ignoring *Inactive elements*	FADDA
FADDV	Floating-point add recursive reduction, treating *Inactive elements* as +0.0	FADDV
FMAXNMV	Floating-point maximum number recursive reduction, treating *Inactive elements* as the default NaN	FMAXNMV
FMAXV	Floating-point maximum recursive reduction, treating *Inactive elements* as negative infinity	FMAXV
FMINNMV	Floating-point minimum number recursive reduction, treating *Inactive elements* as the default NaN	FMINNMV
FMINV	Floating-point minimum recursive reduction, treating *Inactive elements* as positive infinity	FMINV
ORV	Bitwise OR reduction, treating *Inactive elements* as zero	ORV
SADDV	Signed add reduction, treating *Inactive elements* as zero	SADDV
SMAXV	Signed maximum reduction, treating *Inactive elements* as the minimum signed integer	SMAXV
SMINV	Signed minimum reduction, treating *Inactive elements* the maximum signed integer	SMINV
UADDV	Unsigned add reduction, treating *Inactive elements* as zero	UADDV
UMAXV	Unsigned maximum reduction, treating *Inactive elements* as zero	UMAXV
UMINV	Unsigned minimum reduction, treating *Inactive elements* as the maximum unsigned integer	UMINV

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10 Data processing - SVE2

The following subsections describe the SVE2 processing instructions:

- Down-counting loops.
- Constructive multiply.
- Uniform DSP operations.
- Widening DSP operations.
- Narrowing DSP operation.
- Unary narrowing operations.
- Non-widening pairwise arithmetic.
- Widening pairwise arithmetic.
- Bitwise ternary logical instructions.
- Large integer arithmetic.
- Multiplication by indexed elements.
- Complex integer arithmetic.
- Conversions.
- Floating-point widening multiply-accumulate.
- Floating-point integer binary logarithm.
- Cross-lane match detect.
- Bit permutation.
- Polynomial arithmetic.
- Vector permutation.
- Table lookup.
- Cryptography support.
- BFloat16 arithmetic.
- BFloat16 minimum/maximum.
- Floating-point adjust exponent.
- Floating-point multiply-accumulate.
- Dot product.
- Floating-point matrix multiply-accumulate.
- Floating-point round.
- Quadword operations.
- Integer shift and convert.
- Predicate element index.
- Multi-vector predication.
- Permute vector.
- Predicate pair loop control.
- Predicate move.
- Predicate select.

C3.10.1 Down-counting loops

A down-counting `WHILE` instruction decrements the value of the first scalar operand and compares the value with a second, fixed scalar operand. The instruction generates a destination predicate with all of the following characteristics:

- The predicate elements starting from the highest-numbered element are true while the comparison remains true.
- The predicate elements thereafter, down to the lowest-numbered element, are false when the comparison becomes false.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All 32 bits or 64 bits of the scalar operands are significant for the purposes of comparison. The full 32-bit or 64-bit value of the first operand is decremented by 1 for each destination predicate element, irrespective of the element size. The first general-purpose register operand is not updated.

If all of the following occur, a comparison can never fail, resulting in an all-true predicate:

- The comparison includes an equality test.
- The second scalar operand is equal to the minimum integer value of the selected size and type of comparison.

Table C3-191 Down counting loop instructions

Mnemonic	Instruction	See
`WHILEGE`	While decrementing signed 32-bit or 64-bit scalar greater than or equal to scalar	WHILEGE (predicate)
`WHILEGT`	While decrementing signed 32-bit or 64-bit scalar greater than scalar	WHILEGT (predicate)
`WHILEHI`	While decrementing unsigned 32-bit or 64-bit scalar higher than scalar	WHILEHI (predicate)
`WHILEHS`	While decrementing unsigned 32-bit or 64-bit scalar higher or same as scalar	WHILEHS (predicate)

C3.10.2 Constructive multiply

SVE2 includes the following constructive, three-operand versions of the integer multiply instructions:

Table C3-192 Constructive multiply instructions

Mnemonic	Instruction	See
`MUL`	Multiply vectors (unpredicated)	MUL (vectors, unpredicated)
`SMULH`	Signed multiply returning high half (unpredicated)	SMULH (unpredicated)
`UMULH`	Unsigned multiply returning high half (unpredicated)	UMULH (unpredicated)

C3.10.3 Uniform DSP operations

The uniform DSP instructions are based on AArch64 Advanced SIMD instructions with the same mnemonic. The instructions operate on integer operands and produce results with a uniform element size. The operation of an instruction might include one or more of rounding, halving, saturation, and accumulation.

Table C3-193 Uniform DSP instructions

Mnemonic	Instruction	See
`SABA`	Signed absolute difference and accumulate	SABA
`SHADD`	Signed halving addition	SHADD
`SHSUB`	Signed halving subtract	SHSUB
`SHSUBR`	Signed halving subtract reversed vectors	SHSUBR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SLI	Shift left and insert (immediate)	SLI
SQABS	Signed saturating absolute value	SQABS
SQADD	Signed saturating addition (predicated)	SQADD (vectors, predicated)
SQDMULH	Signed saturating doubling multiply high (unpredicated)	SQDMULH (vectors)
SQNEG	Signed saturating negate	SQNEG
SQRDMLAH	Signed saturating rounding doubling multiply-add high to accumulator (unpredicated)	SQRDMLAH (indexed)
SQRDMLSH	Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated)	SQRDMLSH (vectors)
SQRDMULH	Signed saturating rounding doubling multiply high (unpredicated)	SQRDMULH (vectors)
SQRSHL	Signed saturating rounding shift left by vector (predicated)	SQRSHL
SQRSHLR	Signed saturating rounding shift left reversed vectors (predicated)	SQRSHLR
SQSHL	Signed saturating shift left by immediate	SQSHL (immediate)
SQSHL	Signed saturating shift left by vector (predicated)	SQSHL (vectors)
SQSHLR	Signed saturating shift left reversed vectors (predicated)	SQSHLR
SQSHLU	Signed saturating shift left unsigned by immediate	SQSHLU
SQSUB	Signed saturating subtraction (predicated)	SQSUB (vectors, predicated)
SQSUBR	Signed saturating subtraction reversed vectors (predicated)	SQSUBR
SRHADD	Signed rounding halving addition	SRHADD
SRI	Shift right and insert (immediate)	SRI
SRSHL	Signed rounding shift left by vector (predicated)	SRSHL
SRSHLR	Signed rounding shift left reversed vectors (predicated)	SRSHLR
SRSHR	Signed rounding shift right by immediate	SRSHR
SRSRA	Signed rounding shift right and accumulate (immediate)	SRSRA
SSRA	Signed shift right and accumulate (immediate)	SSRA
SUQADD	Signed saturating addition of unsigned value	SUQADD
UABA	Unsigned absolute difference and accumulate.	UABA
UHADD	Unsigned halving addition.	UHADD
UHSUB	Unsigned halving subtract	UHSUB
UHSUBR	Unsigned halving subtract reversed vectors	UHSUBR
UQADD	Unsigned saturating addition (predicated)	UQADD (vectors, predicated)
UQRSHL	Unsigned saturating rounding shift left by vector (predicated)	UQRSHL
UQRSHLR	Unsigned saturating rounding shift left reversed vectors (predicated)	UQRSHLR
UQSHL	Unsigned saturating shift left by immediate	UQSHL (immediate)
UQSHL	Unsigned saturating shift left by vector (predicated)	UQSHL (vectors)
UQSHLR	Unsigned saturating shift left reversed vectors (predicated)	UQSHLR
UQSUB	Unsigned saturating subtraction (predicated)	UQSUB (vectors, predicated)
UQSUBR	Unsigned saturating subtraction reversed vectors (predicated)	UQSUBR
URECPE	Unsigned reciprocal estimate (predicated)	URECPE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
URHADD	Unsigned rounding halving addition.	URHADD
URSHL	Unsigned rounding shift left by vector (predicated)	URSHL
URSHLR	Unsigned rounding shift left reversed vectors (predicated)	URSHLR
URSHR	Unsigned rounding shift right by immediate	URSHR
URSQRTE	Unsigned reciprocal square root estimate (predicated)	URSQRTE
URSRA	Unsigned rounding shift right and accumulate (immediate)	URSRA
USQADD	Unsigned saturating addition of signed value	USQADD
USRA	Unsigned shift right and accumulate (immediate)	USRA

C3.10.4 Widening DSP operations

The widening DSP instructions are based on AArch64 Advanced SIMD instructions with similar mnemonics. The instructions operate on integer values and produce results that are twice the width of some or all of the inputs. The instructions read the narrow inputs from either the even-numbered (bottom) or odd-numbered (top) source elements and place each result in the double-width destination elements that overlap the narrow source elements.

Table C3-194 Widening DSP instructions

Mnemonic	Instruction	See
SABALB	Signed absolute difference and accumulate long (bottom)	SABALB
SABALT	Signed absolute difference and accumulate long (top)	SABALT
SABDLB	Signed absolute difference long (bottom)	SABDLB
SABDLT	Signed absolute difference long (top)	SABDLT
SADDLB	Signed add long (bottom)	SADDLB
SADDLT	Signed add long (top)	SADDLT
SADDWB	Signed add wide (bottom)	SADDWB
SADDWT	Signed add wide (top)	SADDWT
SMLALB	Signed multiply-add long to accumulator (bottom)	SMLALB (vectors)
SMLALT	Signed multiply-add long to accumulator (top)	SMLALT (vectors)
SMLSLB	Signed multiply-subtract long from accumulator (bottom)	SMLSLB (vectors)
SMLSLT	Signed multiply-subtract long from accumulator (top)	SMLSLT (vectors)
SMULLB	Signed multiply long (bottom)	SMULLB (vectors)
SMULLT	Signed multiply long (top)	SMULLT (vectors)
SQDMLALB	Signed saturating doubling multiply-add long to accumulator (bottom)	SQDMLALB (vectors)
SQDMLALT	Signed saturating doubling multiply-add long to accumulator (top)	SQDMLALT (vectors)
SQDMLSLB	Signed saturating doubling multiply-subtract long from accumulator (bottom)	SQDMLSLB (vectors)
SQDMLSLT	Signed saturating doubling multiply-subtract long from accumulator (top)	SQDMLSLT (vectors)
SQDMULLB	Signed saturating doubling multiply long (bottom)	SQDMULLB (vectors)
SQDMULLT	Signed saturating doubling multiply long (top)	SQDMULLT (vectors)
SSHLLB	Signed shift left long by immediate (bottom)	SSHLLB

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SSHLLT	Signed shift left long by immediate (top)	SSHLLT
SSUBLB	Signed subtract long (bottom)	SSUBLB
SSUBLT	Signed subtract long (top)	SSUBLT
SSUBWB	Signed subtract wide (bottom)	SSUBWB
SSUBWT	Signed subtract wide (top)	SSUBWT
UABALB	Unsigned absolute difference and accumulate long (bottom)	UABALB
UABALT	Unsigned absolute difference and accumulate long (top)	UABALT
UABDLB	Unsigned absolute difference long (bottom)	UABDLB
UABDLT	Unsigned absolute difference long (top)	UABDLT
UADDLB	Unsigned add long (bottom)	UADDLB
UADDLT	Unsigned add long (top)	UADDLT
UADDWB	Unsigned add wide (bottom)	UADDWB
UADDWT	Unsigned add wide (top)	UADDWT
UMLALB	Unsigned multiply-add long to accumulator (bottom)	UMLALB (vectors)
UMLALT	Unsigned multiply-add long to accumulator (top)	UMLALT (vectors)
UMLSLB	Unsigned multiply-subtract long from accumulator (bottom)	UMLSLB (vectors)
UMLSLT	Unsigned multiply-subtract long from accumulator (top)	UMLSLT (vectors)
UMULLB	Unsigned multiply long (bottom)	UMULLB (vectors)
UMULLT	Unsigned multiply long (top)	UMULLT (vectors)
USHLLB	Unsigned shift left long by immediate (bottom)	USHLLB
USHLLT	Unsigned shift left long by immediate (top)	USHLLT
USUBLB	Unsigned subtract long (bottom)	USUBLB
USUBLT	Unsigned subtract long (top)	USUBLT
USUBWB	Unsigned subtract wide (bottom)	USUBWB
USUBWT	Unsigned subtract wide (top)	USUBWT

C3.10.5 Narrowing DSP operation

The narrowing DSP instructions are based on AArch64 Advanced SIMD instructions with similar mnemonics. The instructions operate on integer values and produce results that are half the width of the inputs. The instructions read wide source elements and place each result in one of the following:

- The overlapped even-numbered (bottom) half-width destination elements. The odd-numbered destination elements are set to zero.
- The overlapped odd-numbered (top) half-width destination elements. The even-numbered destination elements are unchanged, which means that the instructions are implicitly merging operations.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-195 Narrowing DSP instructions

Mnemonic	Instruction	See
ADDHNB	Add narrow high part (bottom)	ADDHNB
ADDHNT	Add narrow high part (top)	ADDHNT
RADDHNB	Rounding add narrow high part (bottom)	RADDHNB
RADDHNT	Rounding add narrow high part (top)	RADDHNT
RSHRNB	Rounding shift right narrow by immediate (bottom)	RSHRNB
RSHRNT	Rounding shift right narrow by immediate (top)	RSHRNT
RSUBHNB	Rounding subtract narrow high part (bottom)	RSUBHNB
RSUBHNT	Rounding subtract narrow high part (top)	RSUBHNT
SHRNB	Shift right narrow by immediate (bottom)	SHRNB
SHRNT	Shift right narrow by immediate (top)	SHRNT
SQRSHRNB	Signed saturating rounding shift right narrow by immediate (bottom)	SQRSHRNB
SQRSHRNT	Signed saturating rounding shift right narrow by immediate (top)	SQRSHRNT
SQRSHRUNB	Signed saturating rounding shift right unsigned narrow by immediate (bottom)	SQRSHRUNB
SQRSHRUNT	Signed saturating rounding shift right unsigned narrow by immediate (top)	SQRSHRUNT
SQSHRNB	Signed saturating shift right narrow by immediate (bottom)	SQSHRNB
SQSHRNT	Signed saturating shift right narrow by immediate (top)	SQSHRNT
SQSHRUNB	Signed saturating shift right unsigned narrow by immediate (bottom)	SQSHRUNB
SQSHRUNT	Signed saturating shift right unsigned narrow by immediate (top)	SQSHRUNT
SUBHNB	Subtract narrow high part (bottom)	SUBHNB
SUBHNT	Subtract narrow high part (top)	SUBHNT
UQRSHRNB	Unsigned saturating rounding shift right narrow by immediate (bottom)	UQRSHRNB
UQRSHRNT	Unsigned saturating rounding shift right narrow by immediate (top)	UQRSHRNT
UQSHRNB	Unsigned saturating shift right narrow by immediate (bottom)	UQSHRNB
UQSHRNT	Unsigned saturating shift right narrow by immediate (top)	UQSHRNT

C3.10.6 Unary narrowing operations

The unary narrowing instructions are unpredicated and do not write to the source register. The instructions read elements from the source vector and saturate each value to the half-width destination element size. The instructions place the narrow results in one of the following:

- The overlapped even-numbered (bottom) half-width elements. The odd-numbered elements are set to zero.
- The overlapped odd-numbered (top) half-width elements. The even-numbered elements are unchanged.

Non-saturating (truncating) conversions can be performed using existing SVE instructions such as shifts, masks, and permutes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-196 Unary narrowing instructions

Mnemonic	Instruction	See
SQXTNB	Signed saturating extract narrow (bottom)	SQXTNB
SQXTNT	Signed saturating extract narrow (top)	SQXTNT
SQXTUNB	Signed saturating unsigned extract narrow (bottom)	SQXTUNB
SQXTUNT	Signed saturating unsigned extract narrow (top)	SQXTUNT
UQXTNB	Unsigned saturating extract narrow (bottom)	UQXTNB
UQXTNT	Unsigned saturating extract narrow (top)	UQXTNT

C3.10.7 Non-widening pairwise arithmetic

The non-widening pairwise arithmetic instructions operate on pairs of adjacent elements in each source vector and produce a result element that is the same size as a single input element. The results from the first and second source vectors are interleaved, so that the source and result elements overlap. The result is destructively placed in the first source vector. The AArch64 Advanced SIMD instructions do not interleave the results from the first and second source vectors. Predication applies to the destination vector. The even-numbered predicate elements enable an operation on a pair of elements in the first source vector. The odd-numbered predicate elements enable an operation on a pair of elements in the second source vector. *Inactive elements* in the destination vector register are not modified.

Table C3-197 Non-widening pairwise arithmetic instructions

Mnemonic	Instruction	See
ADDP	Add pairwise	ADDP
FADDP	Floating-point add pairwise	FADDP
FMAXNMP	Floating-point maximum number pairwise	FMAXNMP
FMAXP	Floating-point maximum pairwise	FMAXP
FMINNMP	Floating-point minimum number pairwise	FMINNMP
FMINP	Floating-point minimum pairwise	FMINP
SMAXP	Signed maximum pairwise	SMAXP
SMINP	Signed minimum pairwise	SMINP
UMAXP	Unsigned maximum pairwise	UMAXP
UMINP	Unsigned minimum pairwise	UMINP

C3.10.8 Widening pairwise arithmetic

The widening pairwise arithmetic instructions operate on pairs of adjacent elements in a single source vector and produce a double-width result element that is accumulated into the destination vector. *Inactive elements* in the destination vector register are not modified.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-198 Widening pairwise arithmetic instructions

Mnemonic	Instruction	See
SADALP	Signed add and accumulate long pairwise	SADALP
UADALP	Unsigned add and accumulate long pairwise	UADALP

C3.10.9 Bitwise ternary logical instructions

The bitwise ternary logical instructions enable complex bit processing codes to be accelerated using multiple bitwise logical operations in a shorter instruction sequence. All of the following operations are supported by the bitwise ternary logical instructions:

- The BCAX instruction combines a ternary bitwise clear with an exclusive-OR.
- The EOR3 instruction provides a ternary exclusive-OR.
- The XAR instruction combines an exclusive-OR with rotation by a constant amount.
- The bitwise select instructions, BSL, BSL1N, BSL2N, and NBSL, can be used with other bitwise logical instructions to generate all 256 possible bitwise combinations of three input bits using at most three instructions.

The bitwise ternary logical instructions are unpredicated.

Table C3-199 Bitwise ternary logical instructions

Mnemonic	Instruction	See
BCAX	Bitwise clear and exclusive-OR	BCAX
BSL	Bitwise select	BSL
BSL1N	Bitwise select with first input inverted	BSL1N
BSL2N	Bitwise select with second input inverted	BSL2N
EOR3	Bitwise exclusive-OR of three vectors	EOR3
NBSL	Bitwise inverted select	NBSL
XAR	Bitwise exclusive-OR and rotate right by immediate	XAR

C3.10.10 Large integer arithmetic

The large integer arithmetic instructions aid the processing of large integers in vector registers by maintaining multiple carry chains that are interleaved in accumulator vectors.A large integer arithmetic instruction takes as input all of the following:

- Either the even-numbered (bottom) or odd-numbered (top) elements of the first source vector.
- A 1-bit carry input from the least-significant bit of the odd-numbered elements of the second source vector.

The inputs to the instruction are added to or subtracted from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-200 Large integer arithmetic instructions

Mnemonic	Instruction	See
ADCLB	Add with carry long (bottom)	ADCLB
ADCLT	Add with carry long (top)	ADCLT
SBCLB	Subtract with carry long (bottom)	SBCLB
SBCLT	Subtract with carry long (top)	SBCLT

C3.10.11 Multiplication by indexed elements

The multiplication by indexed elements instructions take all integer elements in each 128-bit vector segment of the first source vector and multiplies them by the indexed element in the corresponding segment of the second source vector.

The products might be destructively added to or subtracted from the corresponding elements of an addend vector.

The second source vector elements are specified using an immediate index that selects the same element position in each 128-bit vector segment. The index range is 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the element size.

Table C3-201 Multiplication by indexed elements instructions

Mnemonic	Instruction	See
MLA	Multiply-add to accumulator (indexed)	MLA (indexed)
MLS	Multiply-subtract from accumulator (indexed)	MLS (indexed)
MUL	Multiply (indexed)	MUL (indexed)
SMLALB	Signed multiply-add long to accumulator (bottom, indexed)	SMLALB (indexed)
SMLALT	Signed multiply-add long to accumulator (top, indexed)	SMLALT (indexed)
SMLSLB	Signed multiply-subtract long from accumulator (bottom, indexed)	SMLSLB (indexed)
SMLSLT	Signed multiply-subtract long from accumulator (top, indexed)	SMLSLT (indexed)
SQDMLALT	Signed saturating doubling multiply-add long to accumulator (top, indexed)	SQDMLALT (indexed)
SQDMLSLB	Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed)	SQDMLSLB (indexed)
SQDMLSLT	Signed saturating doubling multiply-subtract long from accumulator (top, indexed)	SQDMLSLB (indexed)
SQDMULLT	Signed saturating doubling multiply long (top, indexed)	SQDMULLT (indexed)
SQRDMLAH	Signed saturating rounding doubling multiply-add high to accumulator (indexed)	SQRDMLAH (indexed)
SQRDMLSH	Signed saturating rounding doubling multiply-subtract high from accumulator (indexed)	SQRDMLSH (indexed)
SQRDMULH	Signed saturating rounding doubling multiply high (indexed)	SQRDMULH (indexed)
UMLALB	Unsigned multiply-add long to accumulator (bottom, indexed)	UMLALB (indexed)
UMLALT	Unsigned multiply-add long to accumulator (top, indexed)	UMLALT (indexed)
UMLSLB	Unsigned multiply-subtract long from accumulator (bottom, indexed)	UMLSLB (indexed)
UMLSLT	Unsigned multiply-subtract long from accumulator (top, indexed)	UMLSLT (indexed)
UMULLB	Unsigned multiply long (bottom, indexed)	UMULLB (indexed)
UMULLT	Unsigned multiply long (top, indexed)	UMULLT (indexed)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.12 Complex integer arithmetic

Complex integer arithmetic instructions operate on signed integer complex numbers in vectors containing the following interleaved element pairs:

- The even-numbered elements contain the real parts of the complex numbers.
- The odd-numbered elements contain the imaginary parts of the complex numbers.

C3.10.12.1 Uniform complex integer arithmetic

The uniform complex integer arithmetic instructions operate on vectors containing integral complex numbers. The instructions operate on complex numbers that are in polar form.

The `CADD` instructions rotate the complex numbers in the second source vector before adding element pairs to the corresponding elements of the first source vector, in a destructive manner. The rotation direction is 90 degrees or 270 degrees from the positive real axis towards the positive imaginary axis.

The `CMLA` instructions transform the operands to enable multiply-add or multiply-subtract operations on complex numbers by combining two of the instructions. The following transformations are done:

- The complex numbers in the second source vector are rotated by 0 degrees or 180 degrees before multiplying by the duplicated real components of the first source vector.
- The complex numbers in the second source vector are rotated by 90 degrees or 270 degrees before multiplying by the duplicated imaginary components of the first source vector.

The resulting products are added to the corresponding components of the destination and addend vector.

Two `CMLA` instructions can be used, as follows:

- `CMLA Zda.S, Zn.S, Zm.S, #A`
- `CMLA Zda.S, Zn.S, Zm.S, #B`

Some meaningful combinations of A and B are:

- A=0, B=90. The complex number vectors, Zn and Zm, are multiplied and the products are added to the complex numbers in Zda.
- A=0, B=270. The complex number conjugates in Zn are multiplied by the complex numbers in Zm and the products are added to the complex numbers in Zda.
- A=180, B=270. The two complex number vectors, Zn and Zm, are multiplied and the products are subtracted from the complex numbers in Zda.
- A=180, B=90. The complex number conjugates in Zn are multiplied by the complex numbers in Zm and the products are subtracted from the complex numbers in Zda.

The `CMLA` indexed form uses a single complex number in each 128-bit segment of the second source vector as the multiplier for all complex numbers in the corresponding first source vector segment. The complex numbers in the second source vector are specified using an immediate index that selects the same complex number position in each 128-bit vector segment. The index range is 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the complex number size.

Table C3-202 Uniform complex integer arithmetic instructions

Mnemonic	Instruction	See
`CADD`	Complex integer add with rotate	CADD
`CMLA`	Complex integer multiply-add with rotate	CMLA (vectors)
	Complex integer multiply-add with rotate (indexed)	CMLA (indexed)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SQCADD	Saturating complex integer add with rotate	SQCADD
SQRDCMLAH	Saturating rounding doubling complex integer multiply-add high with rotate	SQRDCMLAH (vectors)
SQRDCMLAH	Saturating rounding doubling complex integer multiply-add high with rotate (indexed)	SQRDCMLAH (indexed)

C3.10.12.2 Widening complex integer arithmetic

The widening complex integer instructions deinterleave the real and imaginary components of integral complex numbers, and generate complex result components that have a higher numeric precision than the input values. The instructions differ from other complex instructions that process the real and imaginary components of complex numbers and write the complex result components to the destination.

The following instructions are useful when generating the widened components of the result of a complex multiply-add:

- SQDMLALBT, the imaginary results.
- SQDMLSLT, the real results.
- SQDMLALB, the conjugate real results.
- SQDMLSLBT, the conjugate imaginary results.

The following instructions are useful when generating the widened components of the result of a complex addition (X + jY) or (X - jY), given complex numbers X and Y:

- SADDLBT, the imaginary results when computing (X + jY) or real values when computing (X - jY).
- SSUBLBT, the real results when computing (X + jY) or imaginary values when computing (X - jY).

Table C3-203 Widening complex integer arithmetic instructions

Mnemonic	Instruction	See
SADDLBT	Signed add long (bottom + top)	SADDLBT
SQDMLALBT	Signed saturating doubling multiply-add long to accumulator (bottom × top)	SQDMLALBT
SQDMLSLBT	Signed saturating doubling multiply-subtract long from accumulator (bottom × top)	SQDMLSLBT
SSUBLBT	Signed subtract long (bottom - top)	SSUBLBT
SSUBLTB	Signed subtract long (top - bottom)	SSUBLTB

C3.10.12.3 Complex integer dot product

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. The complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector. The wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element that overlaps all four of the elements that form a pair of complex number values in the first source vector.

Each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates four wide real sums or four wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180, or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, considered in polar form, by applying the following transformations before the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector accumulates the imaginary parts of a complex dot product.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the rotation is #180, there is no transformation. The destination vector accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector accumulates the imaginary parts of a complex conjugate dot product.

The indexed form of the instruction selects a single complex number pair within each 128-bit segment of the second source vector to multiply with all complex number pairs within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index which selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment, encoded in 1 or 2 bits depending on the size of the complex number pair.

Each complex number is represented in a vector register as an even and odd pair of elements. The real part is the even-numbered element and the imaginary part is the odd-numbered element.

Table C3-204 Complex integer dot product instructions

Mnemonic	Instruction	See
`CDOT`	Complex integer dot product	CDOT (vectors)
	Complex integer dot product (indexed)	CDOT (indexed)

C3.10.13 Conversions

C3.10.13.1 Floating-point conversions

The floating-point extra conversion instructions convert to and from fully packed vectors of narrower floating-point elements.

The `FCVTLT` instruction converts the top or odd-numbered narrow floating-point vector elements to wider elements of the next higher precision. The conversion is similar to what is done by the widening integer instructions.

The `FCVTNT` and `FCVXNT` instructions convert wider floating-point vector elements to the top or odd-numbered narrower elements of the next lower precision. The conversion is similar to what is done by the narrowing integer instructions.

The `FCVTXNT` and `FCVTX` instructions convert from double-precision to fully packed half-precision in two narrowing steps, double-precision to single-precision and then single-precision to half-precision. The two-step conversion is done without an intermediate rounding error by using von Neumann rounding, which rounds an inexact mantissa to an odd value.

The existing SVE `FCVT` instructions implement the corresponding widening and narrowing conversions on the bottom or even-numbered half-width elements.

Table C3-205 Floating-point extra conversion instructions

Mnemonic	Instruction	See
`FCVTLT`	Floating-point up convert long (top, predicated)	FCVTLT
`FCVTNT`	Floating-point down convert narrow (top, predicated)	FCVTNT (predicated)
`FCVTX`	Floating-point down convert, rounding to odd (predicated)	FCVTX
`FCVTXNT`	Floating-point down convert, rounding to odd (top, predicated)	FCVTXNT

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.13.2 FP8 floating-point conversions

Convert each floating-point element of the source vector while scaling the value, and place the results in the corresponding elements of the destination vector.

For instructions that convert other floating-point formats to FP8 formats, each input element is scaled by adding the value specified in FPMR.NSCALE to its unbounded exponent during the conversion.

For instructions that convert FP8 formats to other floating-point formats, each input element is scaled by subtracting the value specified in FPMR.LSCALE or FPMR.LSCALE2 from its unbounded exponent during the conversion.

For more information, see the instruction descriptions.

Table C3-206 FP8 floating-point convert instructions

Mnemonic	Instruction	See
`BF1CVT, BF2CVT`	8-bit floating-point convert to BFloat16	BF1CVT, BF2CVT
`BF1CVTLT, BF2CVTLT`	8-bit floating-point convert to BFloat16 (top)	BF1CVTLT, BF2CVTLT
`BFCVTN`	BFloat16 convert, narrow and interleave to 8-bit floating-point	BFCVTN
`F1CVT, F2CVT`	8-bit floating-point convert to half-precision	F1CVT, F2CVT
`F1CVTLT, F2CVTLT`	8-bit floating-point convert to half-precision (top)	F1CVTLT, F2CVTLT
`FCVTN`	Half-precision convert, narrow and interleave to 8-bit floating-point	FCVTN
`FCVTNB`	Single-precision convert, narrow and interleave to 8-bit floating-point (bottom)	FCVTNB
`FCVTNT`	Single-precision convert, narrow and interleave to 8-bit floating-point (top)	FCVTNT (unpredicated)

C3.10.14 Multiply-accumulate

C3.10.14.1 Floating-point widening multiply-accumulate

The floating-point widening multiply-accumulate instructions multiply the even-numbered or odd-numbered half-precision elements of the two source vectors and then destructively add or subtract the single-precision intermediate products. Intermediate rounding is not done. The result is placed into the overlapping single-precision elements of the addend vector.

The instructions implicitly convert the half-precision inputs to single-precision and can be used to mitigate the impact of round-off errors when accumulating half-precision floating-point values over many iterations.

The instructions are unpredicated and preserve the multiplier and multiplicand source vectors.

Table C3-207 Floating-point widening multiply-accumulate instructions

Mnemonic	Instruction	See
`FMLALB`	Floating-point fused multiply-add long to accumulator (bottom)	FMLALB (vectors)
`FMLALB`	Floating-point fused multiply-add long to accumulator (bottom, indexed)	FMLALB (indexed)
`FMLALT`	Floating-point fused multiply-add long to accumulator (top)	FMLALT (vectors)
`FMLALT`	Floating-point fused multiply-add long to accumulator (top, indexed)	FMLALT (indexed)
`FMLSLB`	Floating-point fused multiply-subtract long from accumulator (bottom)	FMLSLB (vectors)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FMLSLB	Floating-point fused multiply-subtract long from accumulator (bottom, indexed)	FMLSLB (indexed)
FMLSLT	Floating-point fused multiply-subtract long from accumulator (top)	FMLSLT (vectors)
FMLSLT	Floating-point fused multiply-subtract long from accumulator (top, indexed)	FMLSLT (indexed)

C3.10.14.2 FP8 floating-point widening multiply-accumulate

The FP8 floating-point multiply-accumulate instructions widen the 8-bit floating-point elements in the source vectors to half-precision or single-precision format and multiply the corresponding elements. They generate a fused, downscaled product and accumulate into the corresponding elements of the destination vector, without intermediate rounding.

For these instructions, the unbounded intermediate product is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-208 FP8 floating-point widening multiply-accumulate instructions

Mnemonic	Instruction	See
FMLALB	8-bit floating-point multiply-add long to half-precision (bottom, indexed)	FMLALB (indexed, FP8 to FP16)
	8-bit floating-point multiply-add long to half-precision (bottom)	FMLALB (vectors, FP8 to FP16)
FMLALLBB	8-bit floating-point multiply-add long long to single-precision (bottom bottom, indexed)	FMLALLBB (indexed)
	8-bit floating-point multiply-add long long to single-precision (bottom bottom)	FMLALLBB (vectors)
FMLALLBT	8-bit floating-point multiply-add long long to single-precision (bottom top, indexed)	FMLALLBT (indexed)
	8-bit floating-point multiply-add long long to single-precision (bottom top)	FMLALLBT (vectors)
FMLALLTB	8-bit floating-point multiply-add long long to single-precision (top bottom, indexed)	FMLALLTB (indexed)
	8-bit floating-point multiply-add long long to single-precision (top bottom)	FMLALLTB (vectors)
FMLALLTT	8-bit floating-point multiply-add long long to single-precision (top top, indexed)	FMLALLTT (indexed)
	8-bit floating-point multiply-add long long to single-precision (top top)	FMLALLTT (vectors)
FMLALT	8-bit floating-point multiply-add long to half-precision (top, indexed)	FMLALT (indexed, FP8 to FP16)
	8-bit floating-point multiply-add long to half-precision (top)	FMLALT (vectors, FP8 to FP16)

C3.10.15 Floating-point integer binary logarithm

The floating-point integer binary logarithm instruction returns the signed integer base 2 logarithm of each floating-point input element |x| after normalization.

The instruction produces the unbiased exponent of x used in the representation of the floating-point value. For positive x, $x = \text{significand} \times 2^{\text{exponent}}$.

The integer results are placed in elements of the destination vector, which have the same width as the floating-point input elements:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If x is normal, the result is the base 2 logarithm of x.
- If x is subnormal, the result corresponds to the normalized representation.
- If x is infinite, the result is $2^{(esize-1)}-1$.
- If x is ±0.0 or NaN, the result is $-2^{(esize-1)}$.

Inactive elements in the destination vector register are not modified.

Table C3-209 Floating-point integer binary logarithm instruction

Mnemonic	Instruction	See
`FLOGB`	Floating-point base 2 logarithm as integer	FLOGB

C3.10.16 Floating-point minimum/maximum absolute value

Table C3-210 Floating-point absolute minimum/maximum instructions

Mnemonic	Instruction	See
`FAMAX`	Floating-point absolute maximum (predicated)	FAMAX
`FAMIN`	Floating-point absolute minimum (predicated)	FAMIN

C3.10.17 Cross-lane match detect

This section includes instructions that detect or count matching elements within another vector, or within a 128-bit vector segment.

C3.10.17.1 Vector Histogram Count

The vector histogram count instructions create vector histograms.

- `HISTSEG` compares each 8-bit byte element in the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector. The instruction counts the matching elements and places the result in the corresponding destination vector element. The instruction is unpredicated.
- `HISTCNT` compares each active 32-bit or 64-bit element in the first source vector with all elements in the second source vector that have an element number less than or equal to the *Active element* in the first source vector. The number of matching elements is counted and the result is placed in the corresponding destination vector element. *Inactive elements* in the destination vector are set to zero. *Inactive elements* in the second source vector do not cause a match.

Table C3-211 Vector histogram count instructions

Mnemonic	Instruction	See
`HISTCNT`	Count matching elements in vector	HISTCNT
`HISTSEG`	Count matching elements in vector segments	HISTSEG

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.17.2 Character match

The character match instructions can be used to scan each 128-bit segment of the second source vector for an 8-bit or 16-bit character string from the first source vector.

The `MATCH` and `NMATCH` instructions compare each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. When the first source character matches any (`MATCH`) or does not match any (`NMATCH`) character in the second segment, a true value is placed in the corresponding destination predicate element, otherwise a false value is placed in the destination predicate element. *Inactive elements* in the destination predicate register are set to zero.

The instruction sets the NZC condition flags based on the predicate result, and sets the V flag to zero.

Table C3-212 Character match instructions

Mnemonic	Instruction	See
`MATCH`	Detect any matching elements, setting the condition flag	MATCH
`NMATCH`	Detect no matching elements, setting the condition flags	NMATCH

C3.10.17.3 Contiguous conflict detection

The contiguous conflict detection instructions check two addresses for a conflict or overlap between address ranges that could result in a loop-carried dependency through memory. The address range has the form [addr,addr+*VL*÷8], where *VL* is the Effective SVE vector length in bits. A conflict can occur when contiguous load and store instructions use these addresses within the same loop iteration.

The instructions generate a predicate with elements that are true when the addresses cannot conflict within the same iteration, and false thereafter. The instructions set the NZC condition flags based on the predicate result, and the V flag is set to zero.

Table C3-213 Contiguous conflict detection instructions

Mnemonic	Instruction	See
`WHILERW`	While free of read-after-write conflicts	WHILERW
`WHILEWR`	While free of write-after-read/write conflicts	WHILEWR

C3.10.18 Bit permutation

The bit permutation instructions are optional. The bit permutation instructions are configured by the ID_AA64ZFR0_EL1.BitPerm bit. The instructions can be used to scatter, gather, or separate a set of bits within each first source vector element under the control of a bit mask or sieve in the corresponding second source vector elements. The instructions are unpredicated.

The `BDEP` instruction scatters the lowest-numbered contiguous bits within each first source vector element to the bit positions that are indicated by nonzero bits in the corresponding mask element of the second source vector. The order of the bits is preserved. The bits corresponding to a zero mask bit are set to zero.

The `BEXT` instruction gathers bits in each first source vector element from the bit positions that are indicated by nonzero bits in the corresponding mask element of the second source vector. The bits are gathered to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order. The remaining higher-numbered bits are set to zero.

The `BGRP` instruction selects bits from each first source vector element and groups them into the corresponding destination element, using a corresponding mask element in the second source vector, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The nonzero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the lowest-numbered contiguous bits of the destination element.
- The zero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the highest-numbered contiguous bits of the destination element.

The bit order within each group is preserved.

Table C3-214 Bit permutation instructions

Mnemonic	Instruction	See
BDEP	Scatter lower bits into positions selected by bitmask	BDEP
BEXT	Gather lower bits from positions selected by bitmask	BEXT
BGRP	Group bits to right or left as selected by bitmask	BGRP

C3.10.19 Polynomial arithmetic

The polynomial arithmetic instructions support polynomial arithmetic over [0, 1], where exclusive-OR takes the place of addition. The instructions can be used in applications such as CRC calculations, AES-GCM, elliptic curve cryptography, Diffie-Hellman key exchange, and others.

The PMUL and widening PMLAL, PMULL, PMULLB, and PMULLT instructions perform polynomial multiplication over [0,1]. PMLAL additionally performs a bitwise exclusive-OR of the multiplication result on the element in the destination vector.

The PMULLB and PMULLT instructions read the source operands from either the even-numbered (bottom) or odd-numbered (top) elements, respectively. The widened results are placed in the double-width destination elements that overlap with the corresponding source elements.

The multi-vector PMLAL and PMULL instructions produce results in two destination vectors. The double-width results computed from even-numbered source elements are placed in the first destination vector, while the double-width results computed from odd-numbered source elements are placed in the second destination vector.

The EORBT and EORTB instructions operate on the even-numbered elements of the first source vector register and the odd-numbered elements of the second source vector register. The result is either placed in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged, or placed in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged.

These instructions are unpredicated.

Table C3-215 Polynomial arithmetic instructions

Mnemonic	Instruction	See
EORBT	Interleaving exclusive-OR (bottom, top)	EORBT
EORTB	Interleaving exclusive-OR (top, bottom)	EORTB
PMLAL	Multi-vector polynomial multiply long and accumulate	PMLAL
PMUL	Polynomial multiply vectors (unpredicated)	PMUL
PMULL	Multi-vector polynomial multiply long	PMULL
PMULLB	Polynomial multiply long (bottom)	PMULLB
PMULLT	Polynomial multiply long (top)	PMULLT

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.20 Vector permutation

The vector concatenation instructions `EXT` and `SPLICE` have constructive versions that are introduced in SVE2 that preserve both of the source operands. In the constructive versions of the instruction, only the first source vector register number is encoded, which requires the source vectors to be in consecutively numbered registers (modulo 32).

The `EXPAND` instruction copies the lowest numbered source vector elements to active elements of the destination vector, this complements the `COMPACT` instruction.

For more information see Table C3-185.

Table C3-216 Vector permutation instructions

Mnemonic	Instruction	See
`COMPACT`	Copy active vector elements to lower-numbered elements	COMPACT
`EXPAND`	Copy low-numbered vector elements to active elements	EXPAND
`EXT`	Extract vector from pair of vectors	EXT
`SPLICE`	Splice two vectors under predicate control	SPLICE

C3.10.21 Table lookup

C3.10.21.1 Extended table lookup permute

The SVE2 extended table lookup instructions, `TBL` and `TBX` enable the construction of table lookups or programmable vector permutes where the table consists of two or more vector registers.

Because the index values can select any element in a vector, the instructions are not naturally vector length agnostic.

Table C3-217 Extended table lookup instructions

Mnemonic	Instruction	See
`TBL`	Programmable table lookup in one or two vector table (zeroing)	TBL
`TBX`	Programmable table lookup in single vector table (zeroing)	TBX

C3.10.21.2 Lookup table using 2-bit or 4-bit indices

Copy indexed elements from the first source vector(s) to the destination vector, using packed indices from a portion of the second source vector.

Table C3-218 SVE2 lookup table using 2-bit or 4-bit indices instructions

Mnemonic	Instruction	See
`LUTI2`	Lookup table using 2-bit indices	LUTI2
`LUTI4`	Lookup table using 4-bit indices	LUTI4

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.22 Cryptography support

C3.10.22.1 AES-128 instructions

AES-128 is a 128-bit block cipher that is computed using a combination of linear EOR operations, the use of rotations by fixed values, and a set of 8-bit non-linear substitutions.

The `AESE`, `AESMC`, and `AESEMC` instructions accelerate a single encryption round, and the `AESD`, `AESIMC`, and `AESDIMC` instructions accelerate a single decryption round. These instructions read a 16-byte state from a 128-bit segment of the first source vectors, and a round key from a 128-bit segment of the second source vector.

The AES instructions are unpredicated.

Table C3-219 AES-128 instructions

Mnemonic	Instruction	See
`AESD`	Multi-vector AES single round decryption	AESD (indexed)
	AES single round decryption	AESD (vectors)
`AESDIMC`	Multi-vector AES single round decryption and inverse mix columns	AESDIMC
`AESE`	Multi-vector AES single round encryption	AESE (indexed)
	AES single round encryption	AESE (vectors)
`AESEMC`	Multi-vector AES single round encryption and mix columns	AESEMC
`AESIMC`	AES inverse mix columns	AESIMC
`AESMC`	AES mix columns	AESMC

C3.10.22.2 SHA-3 instructions

The SHA-3 instructions accelerate the SHA-3 hash algorithm.

The SHA-3 hash is based on a running digest of 1600 bits, arranged as a five by five array of 64-bit values. The instructions map the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is done on these registers during a round of the SHA-3 hash calculation.

Two or more parallel SHA-3 hash calculations are combined as a SIMD operation, where one calculation operates on the 0th 64-bit element of each vector, and the other calculation operates on the first 64-bit element of each vector. The SIMD operation is useful for the fast parallel hash algorithm recently introduced into the SHA-3 standard that allows a single input stream to be computed using multiple SHA-3 hashes in parallel.

The SHA3 instructions are unpredicated.

Table C3-220 SHA-3 instructions

Mnemonic	Instruction	See
`RAX1`	Bitwise rotate left by 1 and exclusive-OR	RAX1

See also Bitwise ternary logical instructions.

C3.10.22.3 SM4 instructions

SM4 is the standard Chinese symmetric encryption algorithm which can be accelerated using a similar approach to that used for AES.

SM4 is a 128-bit wide block cipher that is computed using a combination of linear EOR operations, the use of fixed-value rotations, and a set of 8-bit non-linear substitutions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `SM4E` instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, and four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each input data block is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector.
- The `SM4EKEY` instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, and four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector.

The SM4 instructions are unpredicated.

Table C3-221 SM4 instructions

Mnemonic	Instruction	See
`SM4E`	SM4 encryption and decryption	SM4E
`SM4EKEY`	SM4 key updates	SM4EKEY

C3.10.23 BFloat16 arithmetic

C3.10.23.1 Add and subtract

Add or subtract BFloat16 elements of the second source vector from the corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

Table C3-222 SVE2 BFloat16 add and subtract instructions

Mnemonic	Instruction	See
`BFADD`	BFloat16 floating-point add vectors (predicated, unpredicated)	BFADD (predicated) BFADD (unpredicated)
`BFSUB`	BFloat16 floating-point subtract vectors (predicated, unpredicated)	BFSUB (predicated) BFSUB (unpredicated)

C3.10.23.2 Multiply and multiply-accumulate

Multiply all BFloat16 elements of the first source vector by the specified element in the corresponding second source vector and place the results in the corresponding elements of the destination vector. The accumulate instructions add or subtract the result without intermediate rounding from the corresponding elements of the destination vector.

`BFMLA` and `BFMLS` perform a fused multiply-add or multiply-subtract of BFloat16 values without intermediate rounding.

Table C3-223 SVE2 BFloat16 multiply and multiply-accumulate instructions

Mnemonic	Instruction	See
`BFMUL`	BFloat16 floating-point multiply vectors by indexed elements	BFMUL (indexed)
	BFloat16 floating-point multiply vectors (predicated, unpredicated)	BFMUL (vectors, predicated) BFMUL (vectors, unpredicated)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
BFMLA	BFloat16 floating-point fused multiply-add vectors by indexed elements	BFMLA (indexed)
	BFloat16 floating-point fused multiply-add vectors	BFMLA (vectors)
BFMLS	BFloat16 floating-point fused multiply-subtract vectors by indexed elements	BFMLS (indexed)
	BFloat16 floating-point fused multiply-subtract vectors	BFMLS (vectors)

C3.10.24 BFloat16 minimum/maximum

Determine the maximum or minimum active BFloat16 elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

Table C3-224 SVE2 BFloat16 minimum/maximum instructions

Mnemonic	Instruction	See
BFMAX	BFloat16 floating-point maximum (predicated)	BFMAX
BFMAXNM	BFloat16 floating-point maximum number (predicated)	BFMAXNM
BFMIN	BFloat16 floating-point minimum (predicated)	BFMIN
BFMINNM	BFloat16 floating-point minimum number (predicated)	BFMINNM

C3.10.25 Floating-point adjust exponent

The SVE2 BFSCALE instruction performs BFloat16 scaling by integer powers of two.

Table C3-225 SVE2 floating-point adjust exponent instructions

Mnemonic	Instruction	See
BFSCALE	Scales BFloat16 values by integer powers of two	BFSCALE

C3.10.26 Clamp to minimum/maximum

Clamp each element in the destination vectors to between the minimum value in the corresponding element of the first source vector and the maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the destination vector.

C3.10.26.1 Floating-point clamp to minimum/maximum

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-226 SVE2.1 Floating-point clamp to minimum/maximum instructions

Mnemonic	Instruction	See
BFCLAMP	BFloat16 floating-point clamp to minimum/maximum number	BFCLAMP
FCLAMP	Floating-point clamp to minimum/maximum number	FCLAMP

C3.10.26.2 Integer clamp to minimum/maximum

Table C3-227 SVE2.1 Integer clamp to minimum/maximum instructions

Mnemonic	Instruction	See
SCLAMP	Signed clamp to minimum/maximum vector	SCLAMP
UCLAMP	Unsigned clamp to minimum/maximum vector	UCLAMP

C3.10.27 Dot product

C3.10.27.1 Floating-point dot product

Compute the dot product of a pair of values held in the corresponding elements of the first source vectors multiplied by the corresponding elements of the second source vector. The dot product is then destructively added to the corresponding element of the destination vector.

Table C3-228 SVE2.1 Floating-point dot product instructions

Mnemonic	Instruction	See
FDOT	Half-precision floating-point indexed dot product	FDOT (indexed)
	Half-precision floating-point dot product	FDOT (vectors)

C3.10.27.2 FP8 floating-point dot product

The FP8 floating-point dot product instructions accumulate the dot product of two or four adjacent FP8 source elements into half-precision or single-precision destination elements. They generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding.

For these instructions, the unbounded intermediate sum-of-products is scaled by subtracting the value specified in FPMR.LSCALE. For more information, see the instruction descriptions.

Table C3-229 SVE2 FP8 floating-point dot product instructions

Mnemonic	Instruction	See
FDOT	8-bit floating-point indexed dot product to half-precision	FDOT (2-way, indexed, FP8 to FP16)
	8-bit floating-point dot product to half-precision	FDOT (2-way, vectors, FP8 to FP16)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	8-bit floating-point indexed dot product to single-precision	FDOT (4-way, indexed)
	8-bit floating-point dot product to single-precision	FDOT (4-way, vectors)

C3.10.28 Floating-point matrix multiply-accumulate

C3.10.28.1 Floating-point matrix multiply-accumulate

The `FMMLA` (widening, FP16 to FP32) instruction is implemented by FEAT_SVE_F16F32MM. This instruction delimits each source and destination vector into 128-bit segments. The 128-bit segments are organized as follows:

- In the first source vector, a 2x4 matrix of half-precision values in row-by-row order.
- In the second source vector, a 4x2 matrix of half-precision values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

The intermediate single-precision sums-of-products are rounded before they are summed, and the intermediate sum is rounded before addition to the single-precision destination elements.

Table C3-230 SVE2 Floating-point matrix multiply-accumulate instructions

Mnemonic	Instruction	See
`FMMLA`	Half-precision matrix multiply-accumulate to single-precision	FMMLA (widening, half-precision to single-precision)

C3.10.28.2 FP8 floating-point matrix multiply-accumulate

The FP8 `FMMLA` (widening, FP8 to FP32) instruction is implemented by FEAT_F8F32MM. This instruction delimits each source and destination vector into 128-bit segments. The 128-bit segments are organized as follows:

- In the first source vector, a 2x8 matrix of 8-bit floating-point values in row-by-row order.
- In the second source vector, an 8x2 matrix of 8-bit floating-point values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

The FP8 `FMMLA` (widening, FP8 to FP16) instruction is implemented by FEAT_F8F16MM. This instruction delimits each source and destination vector into 64-bit segments. The 64-bit segments are organized as follows:

- In the first source vector, a 2x4 matrix of 8-bit floating-point values in row-by-row order.
- In the second source vector, a 4x2 matrix of 8-bit floating-point values in column-by-column order.
- In the destination vector, a 2x2 matrix of half-precision values in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

These instructions generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding. For these instructions, the intermediate sum of products is downscaled by subtracting the value in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-231 SVE2 FP8 Floating-point matrix multiply-accumulate instructions

Mnemonic	Instruction	See
FMMLA	8-bit floating-point matrix multiply-accumulate to single-precision	FMMLA (widening, FP8 to FP32)
	8-bit floating-point matrix multiply-accumulate to half-precision	FMMLA (widening, FP8 to FP16)

C3.10.29 Floating-point round to integral value

The following instructions round a floating-point value to an integral floating-point value of 32-bit or 64-bit integer size.

These instructions can cause floating-point exceptions, for more information see Floating-point exceptions and exception traps.

Table C3-232 SVE2 Floating-point round to integral value instructions

Mnemonic	Instruction	See
FRINT32X	Floating-point round to 32-bit integer, using current rounding mode (predicated)	FRINT32X
FRINT32Z	Floating-point round to 32-bit integer, towards zero (predicated)	FRINT32Z
FRINT64X	Floating-point round to 64-bit integer, using current rounding mode (predicated)	FRINT64X
FRINT64Z	Floating-point round to 64-bit integer, towards zero (predicated)	FRINT64Z

C3.10.30 Quadword operations

C3.10.30.1 Permute within quadwords

In these instructions the same permutation pattern is applied within each quadword.

Table C3-233 SVE2.1 Permute within quadwords instructions

Mnemonic	Instruction	See
DUPQ	Broadcast indexed element within each quadword vector segment (unpredicated)	DUPQ
EXTQ	Extract vector segment from each pair of quadword vector segments	EXTQ
TBLQ	Programmable table lookup within each quadword vector segment (zeroing)	TBLQ
TBXQ	Programmable table lookup within each quadword vector segment (merging)	TBXQ
UZPQ1	Concatenate even elements within each pair of quadword vector segments	UZPQ1
UZPQ2	Concatenate odd elements within each pair of quadword vector segments	UZPQ2
ZIPQ1	Interleave elements from low halves of each pair of quadword vector segments	ZIPQ1
ZIPQ2	Interleave elements from high halves of each pair of quadword vector segments	ZIPQ2

C3.10.30.2 Reduction to quadword

In the reductions to quadwords instructions a full SVE vector is reduced to a single 128-bit vector. The result of the operations is placed into the corresponding element of the 128-bit Advanced SIMD and floating-point destination register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-234 SVE2.1 Reduction to quadword instructions

Mnemonic	Instruction	See
ADDQV	Unsigned add reduction of quadword vector segments	ADDQV
ANDQV	Bitwise AND reduction of quadword vector segments	ANDQV
EORQV	Bitwise exclusive OR reduction of quadword vector segments	EORQV
FADDQV	Floating-point add recursive reduction of quadword vector segments	FADDQV
FMAXNMQV	Floating-point maximum number recursive reduction of quadword vector segments	FMAXNMQV
FMAXQV	Floating-point maximum reduction of quadword vector segments	FMAXQV
FMINNMQV	Floating-point minimum number recursive reduction of quadword vector segments	FMINNMQV
FMINQV	Floating-point minimum recursive reduction of quadword vector segments	FMINQV
ORQV	Bitwise inclusive OR reduction of quadword vector segments	ORQV
SMAXQV	Signed maximum reduction of quadword vector segments	SMAXQV
SMINQV	Signed minimum reduction of quadword vector segments	SMINQV
UMAXQV	Unsigned maximum reduction of quadword vector segments	UMAXQV
UMINQV	Unsigned minimum reduction of quadword vector segments	UMINQV

C3.10.31 Integer shift and convert

Convert or shift the value of each element of the source vectors and place the results in the corresponding destination vector elements.

These are narrowing, interleaving, saturating variants of the integer shift and convert instructions, to avoid write after write dependencies for merging narrowing top instructions.

Table C3-235 SVE2.1 Integer shift and convert instructions

Mnemonic	Instruction	See
SQCVTN	Signed saturating extract narrow and interleave	SQCVTN
SQCVTUN	Signed saturating unsigned extract narrow and interleave	SQCVTUN
SQRSHRN	Signed saturating rounding shift right narrow by immediate and interleave	SQRSHRN
SQRSHRUN	Signed saturating rounding shift right unsigned narrow by immediate and interleave	SQRSHRUN
UQCVTN	Unsigned saturating extract narrow and interleave	UQCVTN
UQRSHRN	Unsigned saturating rounding shift right narrow by immediate and interleave	UQRSHRN

C3.10.32 Predicate element index

Determine the index of the first or last active TRUE predicate element.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-236 SVE2 Scalar index instructions

Mnemonic	Instruction	See
FIRSTP	Scalar index of first TRUE predicate element (predicated)	FIRSTP
LASTP	Scalar index of last TRUE predicate element (predicated)	LASTP

C3.10.33 Multi-vector predication

Multi-vector predication is required when using the Contiguous multi-vector loads and stores, and also reduces the number of predicate registers and predicate computations in a loop.

The WHILE instructions generate a predicate-as-counter encoding. These instructions have an operand that indicates the number of vectors (2 or 4) to be controlled by this predicate, which determines:

- The maximum value that can be stored in the count.
- The number of elements that are considered *Active* when computing the *Any Active element* and *Last Active element* SVE condition flags.

CNTP has an operand that indicates the limit of the number of elements to be counted. The limit corresponds to the total number of elements in either 2 or 4 vectors.

For more information, see Predicate-as-counter.

Table C3-237 SVE2.1 Multi-vector predication instructions

Mnemonic	Instruction	See
CNTP	Set scalar to count from predicate-as-counter	CNTP (predicate as counter)
PEXT	Set pair of predicates from predicate-as-counter	PEXT (predicate pair)
	Set predicate from predicate-as-counter	PEXT (predicate)
PTRUE	Initialize predicate-as-counter to all active	PTRUE (predicate as counter)
WHILEGE	While decrementing signed scalar greater than or equal to scalar (predicate-as-counter)	WHILEGE (predicate as counter)
WHILEGT	While decrementing signed scalar greater than scalar (predicate-as-counter)	WHILEGT (predicate as counter)
WHILEHI	While decrementing unsigned scalar higher than scalar (predicate-as-counter)	WHILEHI (predicate as counter)
WHILEHS	While decrementing unsigned scalar higher or same as scalar (predicate-as-counter)	WHILEHS (predicate as counter)
WHILELE	While incrementing signed scalar less than or equal to scalar (predicate-as-counter)	WHILELE (predicate as counter)
WHILELO	While incrementing unsigned scalar lower than scalar (predicate-as-counter)	WHILELO (predicate as counter)
WHILELS	While incrementing unsigned scalar lower or same as scalar (predicate-as-counter)	WHILELS (predicate as counter)
WHILELT	While incrementing signed scalar less than scalar (predicate-as-counter)	WHILELT (predicate as counter)

C3.10.34 Permute vector

This instruction is a fixed permute instruction, which is a doubleword variant of the reverse within elements instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-238 SVE2.1 Permute vector instruction

Mnemonic	Instruction	See
REVD	Reverse 64-bit doublewords in elements (predicated)	REVD

C3.10.35 Predicate pair loop control

Generate a pair of predicate registers to simplify loop unrolling.

Table C3-239 SVE2.1 Predicate pair loop control instructions

Mnemonic	Instruction	See
WHILEGE	While decrementing signed scalar greater than or equal to scalar (pair of predicates)	WHILEGE (predicate pair)
WHILEGT	While decrementing signed scalar greater than scalar (pair of predicates)	WHILEGT (predicate pair)
WHILEHI	While decrementing unsigned scalar higher than scalar (pair of predicates)	WHILEHI (predicate pair)
WHILEHS	While decrementing unsigned scalar higher or same as scalar (pair of predicates)	WHILEHS (predicate pair)
WHILELE	While incrementing signed scalar less than or equal to scalar (pair of predicates)	WHILELE (predicate pair)
WHILELO	While incrementing unsigned scalar lower than scalar (pair of predicates)	WHILELO (predicate pair)
WHILELS	While incrementing unsigned scalar lower or same as scalar (pair of predicates)	WHILELS (predicate pair)
WHILELT	While incrementing signed scalar less than scalar (pair of predicates)	WHILELT (predicate pair)

C3.10.36 Predicate move

With the PMOV (to predicate) instruction, based on the specified element size and immediate index, the selected bits in a vector register are expanded into a predicate register ready to be used as an SVE governing predicate.

With the PMOV (to vector) instruction a portion of a bitarray is created and can be stored to memory or further processed in registers.

Table C3-240 SVE2.1 Predicate move instructions

Mnemonic	Instruction	See
PMOV	Move predicate from vector	PMOV (to predicate)
	Move predicate to vector	PMOV (to vector)

C3.10.37 Predicate select

PSEL places contents of the first source predicate register into a destination register or sets the destination to all-FALSE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-241 SVE2.1 Predicate select instruction

Mnemonic	Instruction	See
`PSEL`	Predicate select between predicate register or all-false	PSEL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11 Data processing - SME, SME2

The following subsections describe the SME, SME2, and SME2.1 processing instructions:

- Clamp to minimum/maximum.
- Dot product.
- Vertical dot product.
- Element concatenate and interleave.
- Floating-point conversions.
- Floating-point round.
- Floating-point adjust exponent.
- Lookup table.
- Move operations.
- Multi-vector arithmetic.
- Multi-vector minimum/maximum.
- Multi-vector multiply.
- Multi-vector multiply-accumulate.
- Multi-vector multiply high.
- Multi-vector select.
- Multi-vector shift and convert.
- Outer product.
- Quarter-tile outer product.
- Structured sparsity outer product.
- Stack frame operations.
- Unpack and extend.
- Zero operations.

C3.11.1 Clamp to minimum/maximum

Clamp each element in the destination vectors to between the minimum value in the corresponding element of the first source vector and the maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the destination vectors.

C3.11.1.1 Floating-point clamp to minimum/maximum

Table C3-242 SME2 Floating-point clamp to minimum/maximum instructions

Mnemonic	Instruction	See
`BFCLAMP`	Multi-vector BFloat16 floating-point clamp to minimum/maximum number	BFCLAMP
`FCLAMP`	Multi-vector floating-point clamp to minimum/maximum number	FCLAMP

C3.11.1.2 Integer clamp to minimum/maximum

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-243 SME2 Integer clamp to minimum/maximum instructions

Mnemonic	Instruction	See
SCLAMP	Multi-vector signed clamp to minimum/maximum vector	SCLAMP
UCLAMP	Multi-vector unsigned clamp to minimum/maximum vector	UCLAMP

C3.11.2 Floating-point dot product

Compute the dot product of two or four values held in the corresponding elements of the first source vectors and in the corresponding element of the second source vector. The widened dot product is then added to the corresponding element of the ZA single-vector groups.

C3.11.2.1 BFloat16 floating-point dot product

Table C3-244 BFloat16 floating-point dot product instructions

Mnemonic	Instruction	See
BFDOT	Multi-vector BFloat16 floating-point dot-product by indexed element	BFDOT (multiple and indexed vector)
	Multi-vector BFloat16 floating-point dot-product by vector	BFDOT (multiple and single vector)
	Multi-vector BFloat16 floating-point dot-product	BFDOT (multiple vectors)

C3.11.2.2 Floating-point dot product

Table C3-245 SME2 Floating-point Dot product instructions

Mnemonic	Instruction	See
FDOT	Multi-vector half-precision floating-point dot-product by indexed element	FDOT (multiple and indexed vector)
	Multi-vector half-precision floating-point dot-product by vector	FDOT (multiple and single vector)
	Multi-vector half-precision floating-point dot-product	FDOT (multiple vectors)

C3.11.2.3 FP8 floating-point dot product

The FP8 floating-point dot product instructions accumulate the dot product of two or four FP8 source elements into half-precision or single-precision destination elements. They generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding.

For these instructions, the unbounded intermediate sum-of-products is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-246 SME2 FP8 floating-point convert instructions

Mnemonic	Instruction	See
FDOT	Multi-vector 8-bit floating-point dot-product by indexed element to half-precision	FDOT (2-way, multiple and indexed vector, FP8 to FP16)
	Multi-vector 8-bit floating-point dot-product by vector to half-precision	FDOT (2-way, multiple and single vector, FP8 to FP16)
	Multi-vector 8-bit floating-point dot-product to half-precision	FDOT (2-way, multiple vectors, FP8 to FP16)
	Multi-vector 8-bit floating-point dot-product by indexed element to single-precision	FDOT (4-way, multiple and indexed vector)
	Multi-vector 8-bit floating-point dot-product by vector to single-precision	FDOT (4-way, multiple and single vector)
	Multi-vector 8-bit floating-point dot-product to single-precision	FDOT (4-way, multiple vectors)

C3.11.2.4 Integer dot product

Table C3-247 SME2 Integer Dot product instructions

Mnemonic	Instruction	See
SDOT	Multi-vector signed integer dot-product by indexed element	SDOT (2-way, multiple and indexed vector) SDOT (4-way, multiple and indexed vector)
	Multi-vector signed integer dot-product by vector	SDOT (2-way, multiple and single vector) SDOT (4-way, multiple and single vector)
	Multi-vector signed integer dot-product	SDOT (2-way, multiple vectors) SDOT (4-way, multiple vectors)
SUDOT	Multi-vector signed by unsigned integer dot-product by indexed elements	SUDOT (multiple and indexed vector)
	Multi-vector signed by unsigned integer dot-product by vector	SUDOT (multiple and single vector)
UDOT	Multi-vector unsigned integer dot-product by indexed element	UDOT (2-way, multiple and indexed vector) UDOT (4-way, multiple and indexed vector)
	Multi-vector unsigned integer dot-product by vector	UDOT (2-way, multiple and single vector) UDOT (4-way, multiple and single vector)
	Multi-vector unsigned integer dot-product	UDOT (2-way, multiple vectors) UDOT (4-way, multiple vectors)
USDOT	Multi-vector unsigned by signed integer dot-product by indexed element	USDOT (multiple and indexed vector)
	Multi-vector unsigned by signed integer dot-product by vector	USDOT (multiple and single vector)
	Multi-vector unsigned by signed integer dot-product	USDOT (multiple vectors)

C3.11.3 Vertical dot product

Compute the vertical dot product of the corresponding values held in the first source vectors and in the corresponding elements of second source vector. The widened dot product is then added to the corresponding element of the ZA single-vector groups.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.3.1 BFloat16 vertical dot product

Table C3-248 SME2 BFloat16 floating-point vertical dot product instructions

Mnemonic	Instruction	See
BFVDOT	Multi-vector BFloat16 floating-point vertical dot-product by indexed element	BFVDOT

C3.11.3.2 Floating-point vertical dot product

Table C3-249 SME2 Floating-point vertical dot product instructions

Mnemonic	Instruction	See
FVDOT	Multi-vector half-precision floating-point vertical dot-product by indexed element	FVDOT (FP16 to FP32)

C3.11.3.3 FP8 floating-point vertical dot product

For FP8 vertical dot product instructions, the unbounded intermediate sum-of-products is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-250 SME2 FP8 floating-point vertical dot product instructions

Mnemonic	Instruction	See
FVDOT	Multi-vector 8-bit floating-point vertical dot-product by indexed element to half-precision	FVDOT (FP8 to FP16)
FVDOTB	Multi-vector 8-bit floating-point vertical dot-product by indexed element to single-precision (bottom)	FVDOTB
FVDOTT	Multi-vector 8-bit floating-point vertical dot-product by indexed element to single-precision (top)	FVDOTT

C3.11.3.4 Integer vertical dot product

Table C3-251 SME2 Integer vertical dot product instructions

Mnemonic	Instruction	See
SUVDOT	Multi-vector signed by unsigned integer vertical dot-product by indexed element	SUVDOT
SVDOT	Multi-vector signed integer vertical dot-product by indexed element	SVDOT (2-way) SVDOT (4-way)
USVDOT	Multi-vector unsigned by signed integer vertical dot-product by indexed element	USVDOT
UVDOT	Multi-vector unsigned integer vertical dot-product by indexed element	UVDOT (2-way) UVDOT (4-way)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.4 Element concatenate and interleave

Join elements from the source vectors and place them in the corresponding elements of the destination vectors.

Table C3-252 SME2 Element concatenate/interleave instructions

Mnemonic	Instruction	See
`UZP`	Concatenate elements from four/two vectors	UZP (four registers) UZP (two registers)
`ZIP`	Interleave elements from four/two vectors	ZIP (four registers) ZIP (two registers)

C3.11.5 Floating-point conversions

Convert each element of the source vectors and place the results in the corresponding elements of the destination vectors.

C3.11.5.1 Floating-point conversions

Table C3-253 SME2 Floating-point conversion instructions

Mnemonic	Instruction	See
`FCVT`	Multi-vector floating-point convert from single-precision to packed half-precision	FCVT (narrowing)
	Multi-vector floating-point convert from half-precision to single-precision (in-order)	FCVT (widening)
`FCVTL`	Multi-vector floating-point convert from half-precision to deinterleaved single-precision	FCVTL
`FCVTN`	Multi-vector floating-point convert from single-precision to interleaved half-precision	FCVTN
`FCVTZS`	Multi-vector floating-point convert to signed integer, rounding toward zero	FCVTZS
`FCVTZU`	Multi-vector floating-point convert to unsigned integer, rounding toward zero	FCVTZU

C3.11.5.2 BFloat16 floating-point conversions

Table C3-254 BFloat16 Floating-point conversion instructions

Mnemonic	Instruction	See
`BFCVT`	Multi-vector floating-point convert from single-precision to packed BFloat16 format	BFCVT
`BFCVTN`	Multi-vector floating-point convert from single-precision to interleaved BFloat16 format	BFCVTN

C3.11.5.3 FP8 floating-point conversions

Convert each floating-point element of the source vectors while scaling the value, and place the results in the corresponding elements of the destination vector.

For instructions that convert other floating-point formats to FP8 formats, each input element is scaled by adding the value specified in FPMR.NSCALE to its unbounded exponent during the conversion.

For instructions that convert FP8 formats to other floating-point formats, each input element is scaled by subtracting the value specified in FPMR.LSCALE or FPMR.LSCALE2 from its unbounded exponent during the conversion.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see the instruction descriptions.

Table C3-255 FP8 floating-point conversion instructions

Mnemonic	Instruction	See
`BF1CVT`, `BF2CVT`	Multi-vector floating-point convert from 8-bit floating-point to BFloat16 (in-order)	BF1CVT, BF2CVT
`BF1CVTL`, `BF2CVTL`	Multi-vector floating-point convert from 8-bit floating-point to deinterleaved BFloat16	BF1CVTL, BF2CVTL
`BFCVT`	Multi-vector floating-point convert from BFloat16 to packed 8-bit floating-point format	BFCVT
`F1CVT`, `F2CVT`	Multi-vector floating-point convert from 8-bit floating-point to half-precision (in-order)	F1CVT, F2CVT
`F1CVTL`, `F2CVTL`	Multi-vector floating-point convert from 8-bit floating-point to deinterleaved half-precision	F1CVTL, F2CVTL
`FCVT`	Multi-vector floating-point convert from half-precision to packed 8-bit floating-point format	FCVT (narrowing, FP16 to FP8)
	Multi-vector floating-point convert from single-precision to packed 8-bit floating-point format	FCVT (narrowing, FP32 to FP8)
`FCVTN`	Multi-vector floating-point convert from single-precision to interleaved 8-bit floating-point format	FCVTN (FP32 to FP8)

C3.11.6 Floating-point round

Round each element of the source vectors to an integral floating-point value and place the results in the corresponding elements of the destination vectors.

Table C3-256 SME2 Floating-point round to integral value instructions

Mnemonic	Instruction	See
`FRINTA`	Multi-vector floating-point round to integral value, to nearest with ties away from zero	FRINTA
`FRINTM`	Multi-vector floating-point round to integral value, toward minus Infinity	FRINTM
`FRINTN`	Multi-vector floating-point round to integral value, to nearest with ties to even	FRINTN
`FRINTP`	Multi-vector floating-point round to integral value, toward plus Infinity	FRINTP

C3.11.7 Floating-point adjust exponent

C3.11.7.1 Floating-point scaling

The `FSCALE` instructions perform floating-point scaling by integer powers of two.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-257 SME2 floating-point scaling instructions

Mnemonic	Instruction	See
FSCALE	Multi-vector floating-point adjust exponent	FSCALE (multiple and single vector)
	Multi-vector floating-point adjust exponent	FSCALE (multiple vectors)

C3.11.7.2 BFloat16 scaling

The BFSCALE instructions perform BFloat16 scaling by integer powers of two.

Table C3-258 SME2 BFloat16 scaling instructions

Mnemonic	Instruction	See
BFSCALE	Multi-vector BFloat16 adjust exponent	BFSCALE (multiple and single vector)
	Multi-vector BFloat16 adjust exponent	BFSCALE (multiple vectors)

C3.11.8 Lookup table

Copy elements from ZT0 to the destination vectors using packed indices from a portion of the source vector.

Table C3-259 SME2/SME2.1 Lookup table read instructions

Mnemonic	Instruction	See
LUTI2	Lookup table read using 2-bit indexes	LUTI2 (four registers)
		LUTI2 (single)
		LUTI2 (two registers)
LUTI4	Lookup table read using 4-bit indexes	LUTI4 (four registers, 16-bit & 32-bit)
		LUTI4 (single)
		LUTI4 (two registers)
		LUTI4 (four registers, 8-bit)

C3.11.9 Move operations

Move between ZA single-vector groups, vector registers, or individual horizontal or vertical ZA tile slices.

Table C3-260 SME/SME2 Move instructions

Mnemonic	Instruction	See
MOVA	Move four ZA single-vector groups to four vector registers	MOVA (array to vector, four registers)
	Move two ZA single-vector groups to two vector registers	MOVA (array to vector, two registers)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Move four ZA tile slices to four vector registers	MOVA (tile to vector, four registers)
	Move ZA tile slice to vector register	MOVA (tile to vector, single)
	Move two ZA tile slices to two vector registers	MOVA (tile to vector, two registers)
	Move four vector registers to four ZA single-vector groups	MOVA (vector to array, four registers)
	Move two vector registers to two ZA single-vector groups	MOVA (vector to array, two registers)
	Move four vector registers to four ZA tile slices	MOVA (vector to tile, four registers)
	Move vector register to ZA tile slice	MOVA (vector to tile, single)
	Move two vector registers to two ZA tile slices	MOVA (vector to tile, two registers)

C3.11.9.1 Lookup table move

Move 8 bytes between the ZT0 register and a general-purpose register at a specified byte offset.

`MOVT` (vector to table) copies the source vector register to ZT0 at a specified vector length offset.

Table C3-261 SME2 Lookup table move instructions

Mnemonic	Instruction	See
`MOVT`	Move 8 bytes from general-purpose register to ZT0	MOVT (scalar to table)
	Move 8 bytes from ZT0 to general-purpose register	MOVT (table to scalar)
	Move vector register to ZT0	MOVT (vector to table)

C3.11.9.2 Move and zero

Move contents of ZA single-vector groups or of consecutive horizontal or vertical slices within a ZA tile to the destination vectors and zero the source contents.

Table C3-262 SME2.1 Move and zero instructions

Mnemonic	Instruction	See
`MOVAZ`	Move and zero four ZA single-vector groups to vector registers	MOVAZ (array to vector, four registers)
	Move and zero two ZA single-vector groups to vector registers	MOVAZ (array to vector, two registers)
	Move and zero four ZA tile slices to vector registers	MOVAZ (tile to vector, four registers)
	Move and zero ZA tile slice to vector register	MOVAZ (tile to vector, single)
	Move and zero two ZA tile slices to vector registers	MOVAZ (tile to vector, two registers)

C3.11.10 Multi-vector arithmetic

C3.11.10.1 Add and subtract

Add or subtract vectors from ZA single-vector groups or destructively place results in corresponding elements of the source vectors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-263 SME2 Add and subtract instructions

Mnemonic	Instruction	See
ADD	Add multi-vector to ZA array vector accumulators	ADD (array accumulators)
	Add replicated single vector to multi-vector with ZA array vector results	ADD (array results, multiple and single vector)
	Add multi-vector to multi-vector with ZA array vector results	ADD (array results, multiple vectors)
	Add replicated single vector to multi-vector with multi-vector result	ADD (to vector)
SUB	Subtract multi-vector from ZA array vector accumulators	SUB (array accumulators)
	Subtract replicated single vector from multi-vector with ZA array vector results	SUB (array results, multiple and single vector)
	Subtract multi-vector from multi-vector with ZA array vector results	SUB (array results, multiple vectors)

C3.11.10.2 Add vector to array

Add each element of the source vector to the corresponding Active element of each vertical or horizontal slice of a ZA tile.

Table C3-264 SME Add vector to array instructions

Mnemonic	Instruction	See
ADDHA	Add horizontally vector elements to ZA tile	ADDHA
ADDVA	Add vertically vector elements to ZA tile	ADDVA

C3.11.10.3 Floating-point add and subtract

Destructively add or subtract all elements of the source vectors from the corresponding elements of the ZA single-vector groups.

Table C3-265 SME2 Floating-point add and subtract instructions

Mnemonic	Instruction	See
BFADD	BFloat16 floating-point add multi-vector to ZA array vector accumulators	BFADD
BFSUB	BFloat16 floating-point subtract multi-vector from ZA array vector accumulators	BFSUB
FADD	Floating-point add multi-vector to ZA array vector accumulators	FADD
FSUB	Floating-point subtract multi-vector from ZA array vector accumulators	FSUB

C3.11.11 Multi-vector minimum/maximum

Determine the minimum or maximum of elements of the second source vector and the corresponding elements of the first source vectors and destructively place the results in the corresponding elements of the first source vectors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.11.1 BFloat16 minimum/maximum

These instructions follow the SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors. If FEAT_SVE_B16B16 is implemented, these instructions are also implemented.

Table C3-266 SME2.1 BFloat16 minimum/maximum

Mnemonic	Instruction	See
BFMAX	Multi-vector BFloat16 floating-point maximum by vector	BFMAX (multiple and single vector)
	Multi-vector BFloat16 floating-point maximum	BFMAX (multiple vectors)
BFMAXNM	Multi-vector BFloat16 floating-point maximum number by vector	BFMAXNM (multiple and single vector)
	Multi-vector BFloat16 floating-point maximum number	BFMAXNM (multiple vectors)
BFMIN	Multi-vector BFloat16 floating-point minimum by vector	BFMIN (multiple and single vector)
	Multi-vector BFloat16 floating-point minimum	BFMIN (multiple vectors)
BFMINNM	Multi-vector BFloat16 floating-point minimum number by vector	BFMINNM (multiple and single vector)
	Multi-vector BFloat16 floating-point minimum number	BFMINNM (multiple vectors)

C3.11.11.2 Floating-point minimum/maximum

Table C3-267 SME2 Floating-point minimum/maximum

Mnemonic	Instruction	See
FMAX	Multi-vector floating-point maximum by vector	FMAX (multiple and single vector)
	Multi-vector floating-point maximum	FMAX (multiple vectors)
FMAXNM	Multi-vector floating-point maximum number by vector	FMAXNM (multiple and single vector)
	Multi-vector floating-point maximum number	FMAXNM (multiple vectors)
FMIN	Multi-vector floating-point minimum by vector	FMIN (multiple and single vector)
	Multi-vector floating-point minimum	FMIN (multiple vectors)
FMINNM	Multi-vector floating-point minimum number by vector	FMINNM (multiple and single vectors)
	Multi-vector floating-point minimum number	FMINNM (multiple vectors)

C3.11.11.3 Floating-point minimum/maximum absolute value

Table C3-268 SME2 Floating-point absolute minimum/maximum

Mnemonic	Instruction	See
FAMAX	Multi-vector floating-point absolute maximum	FAMAX
FAMIN	Multi-vector floating-point absolute minimum	FAMIN

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.11.4 Integer minimum/maximum

Table C3-269 SME2 Integer minimum/maximum

Mnemonic	Instruction	See
SMAX	Multi-vector signed maximum by vector	SMAX (multiple and single vector)
	Multi-vector signed maximum	SMAX (multiple vectors)
SMIN	Multi-vector signed minimum by vector	SMIN (multiple and single vector)
	Multi-vector signed minimum	SMIN (multiple vectors)
UMAX	Multi-vector unsigned maximum by vector	UMAX (multiple and single vector)
	Multi-vector unsigned maximum	UMAX (multiple vectors)
UMIN	Multi-vector unsigned minimum by vector	UMIN (multiple and single vector)
	Multi-vector unsigned minimum	UMIN (multiple vectors)

C3.11.12 Multi-vector multiply

Multiply the floating-point elements of the first source vectors with the corresponding elements of the second source vectors and place the results in the corresponding elements of the destination vectors.

C3.11.12.1 BFloat16 multi-vector multiply

Table C3-270 SME2.2 BFloat16 multi-vector multiply instructions

Mnemonic	Instruction	See
BFMUL	Multi-vector BFloat16 multiply by vector	BFMUL (multiple and single vector)
	Multi-vector BFloat16 multiply	BFMUL (multiple vectors)

C3.11.12.2 Floating-point multi-vector multiply

Table C3-271 SME2.2 Floating-point multi-vector multiply instructions

Mnemonic	Instruction	See
FMUL	Multi-vector floating-point multiply by vector	FMUL (multiple and single vector)
	Multi-vector floating-point multiply	FMUL (multiple vectors)

C3.11.13 Multi-vector multiply-accumulate

Multiply the corresponding elements of the first source vectors by the corresponding elements of the second source vector and destructively, without intermediate rounding, add to or subtract from the corresponding elements of the ZA single-vector groups.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.13.1 BFloat16 multiply-accumulate

Table C3-272 SME2 BFloat16 multiply-accumulate instructions

Mnemonic	Instruction	See
BFMLA	Multi-vector BFloat16 floating-point fused multiply-add by indexed element	BFMLA (multiple and indexed vector)
	Multi-vector BFloat16 floating-point fused multiply-add by vector	BFMLA (multiple and single vector)
	Multi-vector BFloat16 floating-point fused multiply-add	BFMLA (multiple vectors)
BFMLAL	Multi-vector BFloat16 floating-point multiply-add long by indexed element	BFMLAL (multiple and indexed vector)
	Multi-vector BFloat16 floating-point multiply-add long by vector	BFMLAL (multiple and single vector)
	Multi-vector BFloat16 floating-point multiply-add long	BFMLAL (multiple vectors)
BFMLS	Multi-vector BFloat16 floating-point fused multiply-subtract by indexed element	BFMLS (multiple and indexed vector)
	Multi-vector BFloat16 floating-point fused multiply-subtract by vector	BFMLS (multiple and single vector)
	Multi-vector BFloat16 floating-point fused multiply-subtract	BFMLS (multiple vectors)
BFMLSL	Multi-vector BFloat16 floating-point multiply-subtract long by indexed element	BFMLSL (multiple and indexed vector)
	Multi-vector BFloat16 floating-point multiply-subtract long by vector	BFMLSL (multiple and single vector)
	Multi-vector BFloat16 floating-point multiply-subtract long	BFMLSL (multiple vectors)

C3.11.13.2 Floating-point multiply-accumulate

Table C3-273 SME2 Floating-point multi-vector multiply-accumulate instructions

Mnemonic	Instruction	See
FMLA	Multi-vector floating-point fused multiply-add by indexed element	FMLA (multiple and indexed vector)
	Multi-vector floating-point fused multiply-add by vector	FMLA (multiple and single vector)
	Multi-vector floating-point fused multiply-add	FMLA (multiple vectors)
FMLAL	Multi-vector floating-point multiply-add long by indexed element	FMLAL (multiple and indexed vector)
	Multi-vector floating-point multiply-add long by vector	FMLAL (multiple and single vector)
	Multi-vector floating-point fused multiply-add long	FMLAL (multiple vectors)
FMLS	Multi-vector floating-point fused multiply-subtract by indexed element	FMLS (multiple and indexed vector)
	Multi-vector floating-point fused multiply-subtract by vector	FMLS (multiple and single vector)
	Multi-vector floating-point fused multiply-subtract	FMLS (multiple vectors)
FMLSL	Multi-vector floating-point multiply-subtract long by indexed element	FMLSL (multiple and indexed vector)
	Multi-vector floating-point multiply-subtract long by vector	FMLSL (multiple and single vector)
	Multi-vector floating-point multiply-subtract long	FMLSL (multiple vectors)

C3.11.13.3 FP8 floating-point widening multiply-accumulate

The FP8 floating-point multiply-accumulate instructions widen the 8-bit floating-point elements in the source vectors to half-precision or single-precision format and multiply the corresponding elements.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

They generate a fused, downscaled product and accumulate into the corresponding elements of the destination vector, without intermediate rounding.

For these instructions, the unbounded intermediate product is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-274 SME2 FP8 floating-point widening multiply-accumulate instructions

Mnemonic	Instruction	See
FMLAL	Multi-vector 8-bit floating-point multiply-add long by indexed element to half-precision	FMLAL (multiple and indexed vector, FP8 to FP16)
	Multi-vector 8-bit floating-point multiply-add long by vector to half-precision	FMLAL (multiple and single vector, FP8 to FP16)
	Multi-vector 8-bit floating-point multiply-add long to half-precision	FMLAL (multiple vectors, FP8 to FP16)
FMLALL	Multi-vector 8-bit floating-point multiply-add long-long by indexed element to single-precision	FMLALL (multiple and indexed vector)
	Multi-vector 8-bit floating-point multiply-add long-long by vector to single-precision	FMLALL (multiple and single vector)
	Multi-vector 8-bit floating-point multiply-add long-long to single-precision	FMLALL (multiple vectors)

C3.11.13.4 Integer multiply-accumulate

Table C3-275 SME2 Integer multi-vector multiply-accumulate instructions

Mnemonic	Instruction	See
SMLAL	Multi-vector signed integer multiply-add long by indexed element	SMLAL (multiple and indexed vector)
	Multi-vector signed integer multiply-add long by vector	SMLAL (multiple and single vector)
	Multi-vector signed integer multiply-add long	SMLAL (multiple vectors)
SMLALL	Multi-vector signed integer multiply-add long-long by indexed element	SMLALL (multiple and indexed vector)
	Multi-vector signed integer multiply-add long-long by vector	SMLALL (multiple and single vector)
	Multi-vector signed integer multiply-add long-long	SMLALL (multiple vectors)
SMLSL	Multi-vector signed integer multiply-subtract long by indexed element	SMLSL (multiple and indexed vector)
	Multi-vector signed integer multiply-subtract long by vector	SMLSL (multiple and single vector)
	Multi-vector signed integer multiply-subtract long	SMLSL (multiple vectors)
SMLSLL	Multi-vector signed integer multiply-subtract long-long by indexed element	SMLSLL (multiple and indexed vector)
	Multi-vector signed integer multiply-subtract long-long by vector	SMLSLL (multiple and single vector)
	Multi-vector signed integer multiply-subtract long-long	SMLSLL (multiple vectors)
SUMLALL	Multi-vector signed by unsigned integer multiply-add long-long by indexed element	SUMLALL (multiple and indexed vector)
	Multi-vector signed by unsigned integer multiply-add long-long by vector	SUMLALL (multiple and single vector)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
UMLAL	Multi-vector unsigned integer multiply-add long by indexed element	UMLAL (multiple and indexed vector)
	Multi-vector unsigned integer multiply-add long by vector	UMLAL (multiple and single vector)
	Multi-vector unsigned integer multiply-add long	UMLAL (multiple vectors)
UMLALL	Multi-vector unsigned integer multiply-add long-long by indexed element	UMLALL (multiple and indexed vector)
	Multi-vector unsigned integer multiply-add long-long by vector	UMLALL (multiple and single vector)
	Multi-vector unsigned integer multiply-add long-long	UMLALL (multiple vectors)
UMLSL	Multi-vector unsigned integer multiply-subtract long by indexed element	UMLSL (multiple and indexed vector)
	Multi-vector unsigned integer multiply-subtract long by vector	UMLSL (multiple and single vector)
	Multi-vector unsigned integer multiply-subtract long by indexed element	UMLSL (multiple vectors)
UMLSLL	Multi-vector unsigned integer multiply-subtract long-long by indexed element	UMLSLL (multiple and indexed vector)
	Multi-vector unsigned integer multiply-subtract long-long by vector	UMLSLL (multiple and single vector)
	Multi-vector unsigned integer multiply-subtract long-long	UMLSLL (multiple vectors)
USMLALL	Multi-vector unsigned by signed integer multiply-add long-long by indexed element	USMLALL (multiple and indexed vector)
	Multi-vector unsigned by signed integer multiply-add long-long by vector	USMLALL (multiple and single vector)
	Multi-vector unsigned by signed integer multiply-add long-long	USMLALL (multiple vectors)

C3.11.14 Multi-vector multiply high

These instructions multiply then double the corresponding elements of the first and second source vectors and destructively place the most significant half of the result in the corresponding elements of the first and second source vectors.

Table C3-276 SME2 Multi-vector multiply high

Mnemonic	Instruction	See
SQDMULH	Multi-vector signed saturating doubling multiply high by vector	SQDMULH (multiple and single vector)
	Multi-vector signed saturating doubling multiply high	SQDMULH (multiple vectors)

C3.11.15 Multi-vector select

Copy into elements of the destination vectors the corresponding elements from the first source vectors where the corresponding predicate elements are TRUE, and from the second source vectors where the corresponding predicate elements are FALSE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-277 SME2 Multi-vector select

Mnemonic	Instruction	See
SEL	Multi-vector conditionally select elements from two vectors	SEL

C3.11.16 Multi-vector shift and convert

Convert or shift the value of each element of the source vectors and place the results in the corresponding destination vector elements.

Table C3-278 SME2 Multi-vector shift and convert instructions

Mnemonic	Instruction	See
SCVTF	Multi-vector signed integer convert to floating-point	SCVTF
SQCVT	Multi-vector signed saturating extract narrow	SQCVT (four registers) SQCVT (two registers)
SQCVTN	Multi-vector signed saturating extract narrow and interleave	SQCVTN
SQCVTU	Multi-vector signed saturating unsigned extract narrow	SQCVTU (four registers) SQCVTU (two registers)
SQCVTUN	Multi-vector signed saturating unsigned extract narrow and interleave	SQCVTUN
SQRSHR	Multi-vector signed saturating rounding shift right narrow by immediate	SQRSHR (four registers) SQRSHR (two registers)
SQRSHRN	Multi-vector signed saturating rounding shift right narrow by immediate and interleave	SQRSHRN
SQRSHRU	Multi-vector signed saturating rounding shift right unsigned narrow by immediate	SQRSHRU (four registers) SQRSHRU (two registers)
SQRSHRUN	Multi-vector signed saturating rounding shift right unsigned narrow by immediate and interleave	SQRSHRUN
SRSHL	Multi-vector signed rounding shift left by vector	SRSHL (multiple and single vector)
	Multi-vector signed rounding shift left	SRSHL (multiple vectors)
UCVTF	Multi-vector unsigned integer convert to floating-point	UCVTF
UQCVT	Multi-vector unsigned saturating extract narrow	UQCVT (four registers) UQCVT (two registers)
UQCVTN	Multi-vector unsigned saturating extract narrow and interleave	UQCVTN
UQRSHR	Multi-vector unsigned saturating rounding shift right narrow by immediate	UQRSHR (four registers) UQRSHR (two registers)
UQRSHRN	Multi-vector unsigned saturating rounding shift right narrow by immediate and interleave	UQRSHRN
URSHL	Multi-vector unsigned rounding shift left by vector	URSHL (multiple and single vector)
	Multi-vector unsigned rounding shift left	URSHL (multiple vectors)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.17 Outer product

Multiply, or widen and multiply, the sub-matrix in the first source vector by the sub-matrix in the second source vector and add or subtract from the destination tile.

C3.11.17.1 BFloat16 outer product

Table C3-279 SME BFloat16 outer product instructions

Mnemonic	Instruction	See
BFMOPA	BFloat16 floating-point outer product and accumulate	BFMOPA (non-widening)
	BFloat16 sum of outer products and accumulate	BFMOPA (widening)
BFMOPS	BFloat16 floating-point outer product and subtract	BFMOPS (non-widening)
	BFloat16 sum of outer products and subtract	BFMOPS (widening)

C3.11.17.2 Binary outer product

Table C3-280 SME2 Binary outer product instructions

Mnemonic	Instruction	See
BMOPA	Bitwise exclusive NOR population count outer product and accumulate	BMOPA
BMOPS	Bitwise exclusive NOR population count outer product and subtract	BMOPS

C3.11.17.3 Floating-point outer product

Table C3-281 SME Floating-point outer product instructions

Mnemonic	Instruction	See
FMOPA	Floating-point outer product and accumulate	FMOPA (non-widening)
	Half-precision floating-point sum of outer products and accumulate	FMOPA (widening)
FMOPS	Floating-point outer product and subtract	FMOPS (non-widening)
	Half-precision floating-point sum of outer products and subtract	FMOPS (widening)

C3.11.17.4 FP8 floating-point outer product

For FP8 outer product instructions, the unbounded intermediate sum of outer products is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-282 SME2 FP8 floating-point outer product instructions

Mnemonic	Instruction	See
FMOPA	8-bit floating-point sum of outer products and accumulate	FMOPA (widening, 2-way, FP8 to FP16)
		FMOPA (widening, 4-way)

C3.11.17.5 Integer outer product

Table C3-283 SME Integer outer product instructions

Mnemonic	Instruction	See
SMOPA	Signed integer sum of outer products and accumulate	SMOPA (2-way) SMOPA (4-way)
SMOPS	Signed integer sum of outer products and subtract	SMOPS (2-way) SMOPS (4-way)
SUMOPA	Signed by unsigned integer sum of outer products and accumulate	SUMOPA
SUMOPS	Signed by unsigned integer sum of outer products and subtract	SUMOPS
UMOPA	Unsigned integer sum of outer products and accumulate	UMOPA (2-way) UMOPA (4-way)
UMOPS	Unsigned integer sum of outer products and subtract	UMOPS (2-way) UMOPS (4-way)
USMOPA	Unsigned by signed integer sum of outer products and accumulate	USMOPA
USMOPS	Unsigned by signed integer sum of outer products and subtract	USMOPS

C3.11.18 Quarter-tile outer product

The quarter-tile outer product instructions generate four independent quarter-tile outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors. The resulting quarter-tile outer products are added to or subtracted from the destination ZA tile.

C3.11.18.1 BFloat16 quarter-tile outer product

Table C3-284 SME BFloat16 quarter-tile outer product instructions

Mnemonic	Instruction	See
BFMOP4A	BFloat16 quarter-tile outer products, accumulating	BFMOP4A (non-widening)
	BFloat16 quarter-tile sums of two outer products, accumulating	BFMOP4A (widening)
BFMOP4S	BFloat16 quarter-tile outer products, subtracting	BFMOP4S (non-widening)
	BFloat16 quarter-tile sums of two outer products, subtracting	BFMOP4S (widening)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.18.2 Floating-point quarter-tile outer product

Table C3-285 SME floating-point quarter-tile outer product instructions

Mnemonic	Instruction	See
FMOP4A	Floating-point quarter-tile outer products, accumulating	FMOP4A (non-widening)
	Half-precision quarter-tile sums of two outer products, accumulating	FMOP4A (widening, 2-way, FP16 to FP32)
FMOP4S	Floating-point quarter-tile outer products, subtracting	FMOP4S (non-widening)
	Half-precision quarter-tile sums of two outer products, subtracting	FMOP4S (widening)

C3.11.18.3 FP8 floating-point quarter-tile outer product

Table C3-286 SME2 FP8 floating-point quarter-tile outer product instructions

Mnemonic	Instruction	See
FMOP4A	8-bit floating-point quarter-tile sums of two outer products, accumulating	FMOP4A (widening, 2-way, FP8 to FP16)
	8-bit floating-point quarter-tile sums of four outer products, accumulating	FMOP4A (widening, 4-way)

C3.11.18.4 Integer quarter-tile outer product

Table C3-287 SME Integer quarter-tile outer product instructions

Mnemonic	Instruction	See
SMOP4A	Signed integer quarter-tile sums of two outer products, accumulating	SMOP4A (2-way)
	Signed integer quarter-tile sums of four outer products, accumulating	SMOP4A (4-way)
SMOP4S	Signed integer quarter-tile sums of two outer products, subtracting	SMOP4S (2-way)
	Signed integer quarter-tile sums of four outer products, subtracting	SMOP4S (4-way)
SUMOP4A	Signed by unsigned integer quarter-tile sums of four outer products, accumulating	SUMOP4A
SUMOP4S	Signed by unsigned integer quarter-tile sums of four outer products, subtracting	SUMOP4S
UMOP4A	Unsigned integer quarter-tile sums of two outer products, accumulating	UMOP4A (2-way)
	Unsigned integer quarter-tile sums of four outer products, accumulating	UMOP4A (4-way)
UMOP4S	Unsigned integer quarter-tile sums of two outer products, subtracting	UMOP4S (2-way)
	Unsigned integer quarter-tile sums of four outer products, subtracting	UMOP4S (4-way)
USMOP4A	Unsigned by signed integer quarter-tile sums of four outer products, accumulating	USMOP4A

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
USMOP4S	Unsigned by signed integer quarter-tile sums of four outer products, subtracting	USMOP4S

C3.11.19 Structured sparsity outer product

The SME structured sparsity outer product instructions accumulate the products or sum-of-products of all elements of the second source vector and selected elements of the first source vectors.

The SME widening structured sparsity outer product instructions widen and multiply a dense matrix by a 2:4 sparse matrix and accumulate to the destination tile. In a 2:4 structured sparsity pattern, two elements are selected per group of four elements, discarding the remaining two elements.

The SME non-widening structured sparsity outer product instructions multiply a dense matrix by a 1:2 sparse matrix and accumulate to the destination tile. In a 1:2 structured sparsity pattern, one element is selected out of a pair of elements, discarding the remaining element.

C3.11.19.1 BFloat16 sparsity outer product

Table C3-288 SME BFloat16 sparsity outer product instructions

Mnemonic	Instruction	See
BFTMOPA	BFloat16 sparse outer product, accumulating	BFTMOPA (non-widening)
	BFloat16 sparse sum of two outer products, accumulating	BFTMOPA (widening)

C3.11.19.2 Floating-point sparsity outer product

Table C3-289 SME Floating-point sparsity outer product instructions

Mnemonic	Instruction	See
FTMOPA	Floating-point sparse outer product, accumulating	FTMOPA (non-widening)
	Half-precision sparse sum of two outer products, accumulating	FTMOPA (widening, 2-way, FP16 to FP32)

C3.11.19.3 FP8 floating-point sparsity outer product

Table C3-290 SME2 FP8 floating-point sparsity outer product instructions

Mnemonic	Instruction	See
FTMOPA	8-bit floating-point sparse sum of two outer products, accumulating	FTMOPA (widening, 2-way, FP8 to FP16)
	8-bit floating-point sparse sum of four outer products, accumulating	FTMOPA (widening, 4-way)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.19.4 Integer sparsity outer product

Table C3-291 SME Integer sparsity outer product instructions

Mnemonic	Instruction	See
STMOPA	Signed integer sparse sum of two outer products, accumulating	STMOPA (2-way)
	Signed integer sparse sum of four outer products, accumulating	STMOPA (4-way)
SUTMOPA	Signed by unsigned integer sparse sum of four outer products, accumulating	SUTMOPA
USTMOPA	Unsigned by signed integer sparse sum of four outer products, accumulating	USTMOPA
UTMOPA	Unsigned integer sparse sum of two outer products, accumulating	UTMOPA (2-way)
	Unsigned integer sparse sum of four outer products, accumulating	UTMOPA (4-way)

C3.11.20 Stack frame operations

Multiply the Streaming SVE vector register size in bytes by an immediate and add the result to the destination general-purpose register or current stack pointer. RDSVL only places the result in the destination general-purpose register without addition.

Table C3-292 SME Stack frame instructions

Mnemonic	Instruction	See
ADDSPL	Add multiple of Streaming SVE predicate register size to scalar register	ADDSPL
ADDSVL	Add multiple of Streaming SVE vector register size to scalar register	ADDSVL
RDSVL	Read multiple of Streaming SVE vector register size to scalar register	RDSVL

C3.11.21 Unpack and extend

Unpack elements from the source vectors and extend them to place in elements of twice their size within the destination vectors.

Table C3-293 SME2 Unpack and extend instructions

Mnemonic	Instruction	See
SUNPK	Unpack and sign-extend multi-vector elements	SUNPK
UUNPK	Unpack and zero-extend multi-vector elements	UUNPK

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.22 Zero operations

Set all corresponding bytes or ZA vector groups to zero.

Table C3-294 SME/SME2 Zero instructions

Mnemonic	Instruction	See
ZERO	Zero ZT0	ZERO (table)
	Zero a list of 64-bit element ZA tiles	ZERO (tiles)

C3.11.22.1 Zero ZA array vectors

Table C3-295 SME2.1 Zero ZA array vectors instructions

Mnemonic	Instruction	See
ZERO	Zero ZA double-vector groups	ZERO (double-vector)
	Zero ZA quad-vector groups	ZERO (quad-vector)
	Zero ZA single-vector groups	ZERO (single-vector)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C4
A64 Instruction Set Encoding

This chapter describes the encoding of the A64 instruction set. It contains the following section:

- A64 instruction set encoding.

In this chapter:

- In the decode tables, an entry of `x` for a field value means the value of the field does not affect the decoding.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1 A64 instruction set encoding

31	30 29	28 25	24 0
op0		op1	

op0	op1	Instruction details
0	0000	Reserved
1	0000	SME encodings
x	0010	SVE encodings
x	00x1	UNALLOCATED
x	100x	Data Processing – Immediate
x	101x	Branches, Exception Generating and System instructions
x	x101	Data Processing – Register
x	x111	Data Processing – Scalar Floating-Point and Advanced SIMD
x	x1x0	Loads and Stores

Reserved

Reserved

The encodings in this section are decoded from A64 instruction set encoding.

31	30 29	28	27	26	25	24 16	15 0
0	op0	0	0	0	0	op1	

op0	op1	Instruction details
00	000000000	UDF
00	!= 000000000	UNALLOCATED
!= 00	xxxxxxxxx	UNALLOCATED

SME encodings

SME encodings

The encodings in this section are decoded from A64 instruction set encoding.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30 29	28 25	24 10	9 6	5 0
1	op0	0 0 0 0	op1		op2

op0	op1	op2	Instruction details
00	x00xxxxx0x00000	0xx0xx	SME2 Quarter Tile Outer Product - 16-bit and 32-bit
00	x00xxxxx0x00000	1xx0xx	UNALLOCATED
00	x00xxxxx0x00001	xxx0xx	UNALLOCATED
00	x00xxxxx0x0001x	xxx0xx	UNALLOCATED
00	x00xxxxx0x001xx	xxx0xx	UNALLOCATED
00	x00xxxxx0x01xxx	xxx0xx	UNALLOCATED
00	x00xxxxx1x0xxxx	xxx0xx	UNALLOCATED
00	x01xxxxxxx0xxxx	xxx0xx	SME2 Sparse Outer Product
00	x0xxxxxxxx0xxxx	xxx1xx	UNALLOCATED
00	x0xxxxxxxx1xxxx	xxxxxx	UNALLOCATED
00	x10xxxxxxxxxxxx	xx00xx	SME FP Outer Product - 32 bit
00	x10xxxxxxxxxxxx	xx10xx	SME2 Outer Product - Misc
01	00xxxxxxxxxxxxx	xxxxxx	SME2 Multi-vector - Memory (Contiguous)
01	10xxxxxxxxxxxxx	xxxxxx	SME2 Multi-vector - Memory (Strided)
01	x10xxxxxxxxxxxx	xxx0xx	SME Integer Outer Product - 32 bit
0x	x10xxxxxxxxxxxx	xxx1xx	UNALLOCATED
0x	x11xxxxx0000000	0x1xxx	SME2 Quarter Tile Outer Product - 64-bit
0x	x11xxxxx0000000	1x1xxx	UNALLOCATED
0x	x11xxxxx0000001	xx1xxx	UNALLOCATED
0x	x11xxxxx000001x	xx1xxx	UNALLOCATED
0x	x11xxxxx00001xx	xx1xxx	UNALLOCATED
0x	x11xxxxx0001xxx	xx1xxx	UNALLOCATED
0x	x11xxxxx001xxxx	xx1xxx	UNALLOCATED
0x	x11xxxxx01xxxxx	xx1xxx	UNALLOCATED
0x	x11xxxxx1xxxxxx	xx1xxx	UNALLOCATED
0x	x11xxxxxxxxxxxx	xx0xxx	SME Outer Product - 64 bit
10	0000010xxxxxxxx	xxxxxx	SME zero array
10	0000011xxxxxxxx	xxxxxx	SME2 Multiple Zero
10	0010010xxxxxxxx	xxxxxx	SME2 zero lookup table
10	0010011xxxxxxxx	xxxxxx	SME2 Move Lookup Table
10	00x011xxxxxxxxx	xxxxxx	UNALLOCATED
10	010011xxxxxxxxx	xxxxxx	SME2 Expand Lookup Table (Non-contiguous)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
10	011011xxxxxxxxx	xxxxxx	UNALLOCATED
10	01x001xxxxxxxxx	xxxxxx	SME2 Expand Lookup Table (Contiguous)
10	0xx000x0xxxxxxx	x0xxxx	SME Move into Array
10	0xx000x0xxxxxxx	x1xxxx	UNALLOCATED
10	0xx000x1xxxxxxx	xxxxxx	SME Move from Array
10	0xx010xxxxxxxxx	xx0xxx	SME Add Vector to Array
10	0xx010xxxxxxxxx	xx1xxx	UNALLOCATED
10	0xx1xxxxxxxxxxx	xxxxxx	UNALLOCATED
10	10x10xxxx0xxxxx	xxxxxx	SME2 Multi-vector - Multiple and Single Array Vectors (Two registers)
10	10x11xxxx0xxxxx	xxxxxx	SME2 Multi-vector - Multiple and Single Array Vectors (Four registers)
10	11x1xxxx00xxxxx	xxxxxx	SME2 Multi-vector - Multiple Array Vectors (Two registers)
10	11x1xxxx10xxxxx	xxxxxx	SME2 Multi-vector - Multiple Array Vectors (Four registers)
10	1xx00xxxxxxxxxx	xxxxxx	SME2 Multi-vector - Indexed (One register)
10	1xx01xxxx0xxxxx	xxxxxx	SME2 Multi-vector - Indexed (Two registers)
10	1xx01xxxx1xxxxx	xxxxxx	SME2 Multi-vector - Indexed (Four registers)
10	1xx10xxxx10100x	xxxxxx	SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers)
10	1xx10xxxx10101x	xxxx0x	SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers)
10	1xx10xxxx10101x	xxxx1x	UNALLOCATED
10	1xx11xxxx1010xx	xxxxxx	UNALLOCATED
10	1xx1xxx00101110	xxxx0x	SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers)
10	1xx1xxx00101111	xxxx0x	SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers)
10	1xx1xxx0010111x	xxxx1x	UNALLOCATED
10	1xx1xxx1010111x	xxxxxx	UNALLOCATED
10	1xx1xxxx0101100	xxxxxx	SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers)
10	1xx1xxxx0101101	xxxxxx	SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers)
10	1xx1xxxx01111xx	xxxxxx	UNALLOCATED
10	1xx1xxxx11x11xx	xxxxxx	UNALLOCATED
10	1xx1xxxxx100xxx	xxxxxx	SME2 Multi-vector - SVE Select
10	1xx1xxxxx110xxx	xxxxxx	SME2 Multi-vector - SVE Constructive Binary
10	1xx1xxxxx111000	xxxxxx	SME2 Multi-vector - SVE Constructive Unary

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
10	1xx1xxxxx111001	0xxxx0	SME2 Multi-vector - FP Multiply
10	1xx1xxxxx111001	0xxxx1	UNALLOCATED
10	1xx1xxxxx111001	1xxxxx	UNALLOCATED
10	1xx1xxxxx111010	0xxxx0	SME2 Multiple and Single Vector - FP multiply
10	1xx1xxxxx111010	0xxxx1	UNALLOCATED
10	1xx1xxxxx111010	1xxxxx	UNALLOCATED
10	1xx1xxxxx111011	xxxxxx	UNALLOCATED
11	xxxxxxxxxxxxxxx	xxxxxx	SME Memory

SME2 Quarter Tile Outer Product - 16-bit and 32-bit

SME2 Quarter Tile Outer Product - 16-bit and 32-bit

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20	17	16	15	14				10	9	6	5	4	3	2	1	0
1	0	0	0	0	0	0	op0	0	0	op1			0	op2	0	0	0	0	0			0	op3		0	op4	

op0	op1	op2	op3	op4	Instruction details
0	0	0	x0	x	SME2 FP32 non-widening quarter tile outer product
0	0	0	x1	0	UNALLOCATED
0	1	0	00	x	SME2 FP8 to FP32 quarter tile outer product
0	1	0	01	0	SME2 FP8 to FP16 quarter tile outer product
0	1	0	10	1	UNALLOCATED
0	1	0	1x	0	UNALLOCATED
1	0	0	x0	x	SME2 BF16 to FP32 quarter tile outer product
1	0	0	x1	0	SME2 FP16 non-widening quarter tile outer product
1	1	0	x0	x	SME2 FP16 to FP32 quarter tile outer product
1	1	0	x1	0	SME2 BF16 non-widening quarter tile outer product
x	0	1	x1	x	SME2 Int16 to Int32 quarter tile outer product
x	1	1	x1	x	UNALLOCATED
x	x	0	x1	1	UNALLOCATED
x	x	1	x0	x	SME2 Int8 to Int32 quarter tile outer product

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 FP32 non-widening quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	S	0	0	ZAda	

M	N	S	Instruction Details	Feature
0	0	0	FMOP4A (non-widening) — Single-precision, single vectors	FEAT_SME_MOP4
0	0	1	FMOP4S (non-widening) — Single-precision, single vectors	FEAT_SME_MOP4
0	1	0	FMOP4A (non-widening) — Single-precision, multiple and single vectors	FEAT_SME_MOP4
0	1	1	FMOP4S (non-widening) — Single-precision, multiple and single vectors	FEAT_SME_MOP4
1	0	0	FMOP4A (non-widening) — Single-precision, single and multiple vectors	FEAT_SME_MOP4
1	0	1	FMOP4S (non-widening) — Single-precision, single and multiple vectors	FEAT_SME_MOP4
1	1	0	FMOP4A (non-widening) — Single-precision, multiple vectors	FEAT_SME_MOP4
1	1	1	FMOP4S (non-widening) — Single-precision, multiple vectors	FEAT_SME_MOP4

SME2 FP8 to FP32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	0	0	0	ZAda	

M	N	Instruction Details	Feature
0	0	FMOP4A (widening, 4-way) — Single vectors	FEAT_SME_MOP4 && FEAT_SME_F8F32
0	1	FMOP4A (widening, 4-way) — Multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F8F32

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	N	Instruction Details	Feature
1	0	FMOP4A (widening, 4-way) — Single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F8F32
1	1	FMOP4A (widening, 4-way) — Multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F8F32

SME2 FP8 to FP16 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	0	1	0	0	ZA

M	N	Instruction Details	Feature
0	0	FMOP4A (widening, 2-way, FP8 to FP16) — Single vectors	FEAT_SME_MOP4 && FEAT_SME_F8F16
0	1	FMOP4A (widening, 2-way, FP8 to FP16) — Multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F8F16
1	0	FMOP4A (widening, 2-way, FP8 to FP16) — Single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F8F16
1	1	FMOP4A (widening, 2-way, FP8 to FP16) — Multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F8F16

SME2 BF16 to FP32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	S	0	0	ZAda	

M	N	S	Instruction Details	Feature
0	0	0	BFMOP4A (widening) — Single vectors	FEAT_SME_MOP4
0	0	1	BFMOP4S (widening) — Single vectors	FEAT_SME_MOP4
0	1	0	BFMOP4A (widening) — Multiple and single vectors	FEAT_SME_MOP4
0	1	1	BFMOP4S (widening) — Multiple and single vectors	FEAT_SME_MOP4

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	N	S	Instruction Details	Feature
1	0	0	BFMOP4A (widening) — Single and multiple vectors	FEAT_SME_MOP4
1	0	1	BFMOP4S (widening) — Single and multiple vectors	FEAT_SME_MOP4
1	1	0	BFMOP4A (widening) — Multiple vectors	FEAT_SME_MOP4
1	1	1	BFMOP4S (widening) — Multiple vectors	FEAT_SME_MOP4

SME2 FP16 non-widening quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19	17	16	15	14					10	9	8	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	M	Zm		0	0	0	0	0	0	0		N	Zn		0	S	1	0	0	ZA

M	N	S	Instruction Details	Feature
0	0	0	FMOP4A (non-widening) — Half-precision, single vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
0	0	1	FMOP4S (non-widening) — Half-precision, single vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
0	1	0	FMOP4A (non-widening) — Half-precision, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
0	1	1	FMOP4S (non-widening) — Half-precision, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
1	0	0	FMOP4A (non-widening) — Half-precision, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
1	0	1	FMOP4S (non-widening) — Half-precision, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
1	1	0	FMOP4A (non-widening) — Half-precision, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
1	1	1	FMOP4S (non-widening) — Half-precision, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16

SME2 FP16 to FP32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19	17	16	15	14					10	9	8	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	M	Zm		0	0	0	0	0	0	0		N	Zn		0	S	0	0	ZAda	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	N	S	Instruction Details	Feature
0	0	0	FMOP4A (widening, 2-way, FP16 to FP32) — Single vectors	FEAT_SME_MOP4
0	0	1	FMOP4S (widening) — Single vectors	FEAT_SME_MOP4
0	1	0	FMOP4A (widening, 2-way, FP16 to FP32) — Multiple and single vectors	FEAT_SME_MOP4
0	1	1	FMOP4S (widening) — Multiple and single vectors	FEAT_SME_MOP4
1	0	0	FMOP4A (widening, 2-way, FP16 to FP32) — Single and multiple vectors	FEAT_SME_MOP4
1	0	1	FMOP4S (widening) — Single and multiple vectors	FEAT_SME_MOP4
1	1	0	FMOP4A (widening, 2-way, FP16 to FP32) — Multiple vectors	FEAT_SME_MOP4
1	1	1	FMOP4S (widening) — Multiple vectors	FEAT_SME_MOP4

SME2 BF16 non-widening quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	S	1	0	0	ZA

M	N	S	Instruction Details	Feature
0	0	0	BFMOP4A (non-widening) — Single vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
0	0	1	BFMOP4S (non-widening) — Single vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
0	1	0	BFMOP4A (non-widening) — Multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
0	1	1	BFMOP4S (non-widening) — Multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
1	0	0	BFMOP4A (non-widening) — Single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
1	0	1	BFMOP4S (non-widening) — Single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
1	1	0	BFMOP4A (non-widening) — Multiple vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
1	1	1	BFMOP4S (non-widening) — Multiple vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 Int16 to Int32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19	17	16	15	14					10	9	8	6	5	4	3	2	1	0
1	0	0	0	0	0	0	u0	0	0	0	M	Zm		0	1	0	0	0	0	0		N	Zn		0	S	1	0	ZAda	

u0	M	N	S	Instruction Details	Feature
0	0	0	0	SMOP4A (2-way) — Single vectors	FEAT_SME_MOP4
0	0	0	1	SMOP4S (2-way) — Single vectors	FEAT_SME_MOP4
0	0	1	0	SMOP4A (2-way) — Multiple and single vectors	FEAT_SME_MOP4
0	0	1	1	SMOP4S (2-way) — Multiple and single vectors	FEAT_SME_MOP4
0	1	0	0	SMOP4A (2-way) — Single and multiple vectors	FEAT_SME_MOP4
0	1	0	1	SMOP4S (2-way) — Single and multiple vectors	FEAT_SME_MOP4
0	1	1	0	SMOP4A (2-way) — Multiple vectors	FEAT_SME_MOP4
0	1	1	1	SMOP4S (2-way) — Multiple vectors	FEAT_SME_MOP4
1	0	0	0	UMOP4A (2-way) — Single vectors	FEAT_SME_MOP4
1	0	0	1	UMOP4S (2-way) — Single vectors	FEAT_SME_MOP4
1	0	1	0	UMOP4A (2-way) — Multiple and single vectors	FEAT_SME_MOP4
1	0	1	1	UMOP4S (2-way) — Multiple and single vectors	FEAT_SME_MOP4
1	1	0	0	UMOP4A (2-way) — Single and multiple vectors	FEAT_SME_MOP4
1	1	0	1	UMOP4S (2-way) — Single and multiple vectors	FEAT_SME_MOP4
1	1	1	0	UMOP4A (2-way) — Multiple vectors	FEAT_SME_MOP4
1	1	1	1	UMOP4S (2-way) — Multiple vectors	FEAT_SME_MOP4

SME2 Int8 to Int32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19	17	16	15	14					10	9	8	6	5	4	3	2	1	0
1	0	0	0	0	0	0	u0	0	0	u1	M	Zm		0	1	0	0	0	0	0		N	Zn		0	S	0	0	ZAda	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	u1	M	N	S	Instruction Details	Feature
0	0	0	0	0	SMOP4A (4-way) — 32-bit, single vectors	FEAT_SME_MOP4
0	0	0	0	1	SMOP4S (4-way) — 32-bit, single vectors	FEAT_SME_MOP4
0	0	0	1	0	SMOP4A (4-way) — 32-bit, multiple and single vectors	FEAT_SME_MOP4
0	0	0	1	1	SMOP4S (4-way) — 32-bit, multiple and single vectors	FEAT_SME_MOP4
0	0	1	0	0	SMOP4A (4-way) — 32-bit, single and multiple vectors	FEAT_SME_MOP4
0	0	1	0	1	SMOP4S (4-way) — 32-bit, single and multiple vectors	FEAT_SME_MOP4
0	0	1	1	0	SMOP4A (4-way) — 32-bit, multiple vectors	FEAT_SME_MOP4
0	0	1	1	1	SMOP4S (4-way) — 32-bit, multiple vectors	FEAT_SME_MOP4
0	1	0	0	0	SUMOP4A — 32-bit, single vectors	FEAT_SME_MOP4
0	1	0	0	1	SUMOP4S — 32-bit, single vectors	FEAT_SME_MOP4
0	1	0	1	0	SUMOP4A — 32-bit, multiple and single vectors	FEAT_SME_MOP4
0	1	0	1	1	SUMOP4S — 32-bit, multiple and single vectors	FEAT_SME_MOP4
0	1	1	0	0	SUMOP4A — 32-bit, single and multiple vectors	FEAT_SME_MOP4
0	1	1	0	1	SUMOP4S — 32-bit, single and multiple vectors	FEAT_SME_MOP4
0	1	1	1	0	SUMOP4A — 32-bit, multiple vectors	FEAT_SME_MOP4
0	1	1	1	1	SUMOP4S — 32-bit, multiple vectors	FEAT_SME_MOP4
1	0	0	0	0	USMOP4A — 32-bit, single vectors	FEAT_SME_MOP4
1	0	0	0	1	USMOP4S — 32-bit, single vectors	FEAT_SME_MOP4
1	0	0	1	0	USMOP4A — 32-bit, multiple and single vectors	FEAT_SME_MOP4
1	0	0	1	1	USMOP4S — 32-bit, multiple and single vectors	FEAT_SME_MOP4
1	0	1	0	0	USMOP4A — 32-bit, single and multiple vectors	FEAT_SME_MOP4
1	0	1	0	1	USMOP4S — 32-bit, single and multiple vectors	FEAT_SME_MOP4
1	0	1	1	0	USMOP4A — 32-bit, multiple vectors	FEAT_SME_MOP4
1	0	1	1	1	USMOP4S — 32-bit, multiple vectors	FEAT_SME_MOP4
1	1	0	0	0	UMOP4A (4-way) — 32-bit, single vectors	FEAT_SME_MOP4
1	1	0	0	1	UMOP4S (4-way) — 32-bit, single vectors	FEAT_SME_MOP4
1	1	0	1	0	UMOP4A (4-way) — 32-bit, multiple and single vectors	FEAT_SME_MOP4
1	1	0	1	1	UMOP4S (4-way) — 32-bit, multiple and single vectors	FEAT_SME_MOP4

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	u1	M	N	S	Instruction Details	Feature
1	1	1	0	0	UMOP4A (4-way) — 32-bit, single and multiple vectors	FEAT_SME_MOP4
1	1	1	0	1	UMOP4S (4-way) — 32-bit, single and multiple vectors	FEAT_SME_MOP4
1	1	1	1	0	UMOP4A (4-way) — 32-bit, multiple vectors	FEAT_SME_MOP4
1	1	1	1	1	UMOP4S (4-way) — 32-bit, multiple vectors	FEAT_SME_MOP4

SME2 Sparse Outer Product

SME2 Sparse Outer Product

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20	16	15	14	13	12	4	3	2	1	0
1	0	0	0	0	0	0	op0	0	1	op1			op2	0	op3			op4	0	op5	

op0	op1	op2	op3	op4	op5	Instruction details	Feature
0	0	0	0	0	x	FTMOPA (non-widening) — Single-precision	FEAT_SME_TMOP
0	0	0	0	1	0	UNALLOCATED	-
0	1	0	0	0	x	FTMOPA (widening, 4-way)	FEAT_SME_TMOP && FEAT_SME_F8F32
0	1	0	0	1	0	FTMOPA (widening, 2-way, FP8 to FP16)	FEAT_SME_TMOP && FEAT_SME_F8F16
1	0	0	0	0	x	BFTMOPA (widening)	FEAT_SME_TMOP
1	0	0	0	1	0	FTMOPA (non-widening) — Half-precision	FEAT_SME_TMOP && FEAT_SME_F16F16
1	1	0	0	0	x	FTMOPA (widening, 2-way, FP16 to FP32)	FEAT_SME_TMOP
1	1	0	0	1	0	BFTMOPA (non-widening)	FEAT_SME_TMOP && FEAT_SME_B16B16
x	0	1	0	1	x	SME2 Int16 to Int32 sparse outer product	-
x	1	1	0	1	x	UNALLOCATED	-
x	x	0	0	1	1	UNALLOCATED	-
x	x	1	0	0	x	SME Int8 to Int32 sparse outer product	-
x	x	x	1	x	x	UNALLOCATED	-

SME2 Int16 to Int32 sparse outer product

The encodings in this section are decoded from SME2 Sparse Outer Product.

31						25	24	23	22	21	20	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	0	0	0	0	0	0	u0	0	1	0	Zm		1	0	0	K	Zk		Zn		i2		1	0	ZAda	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	Instruction Details	Feature
0	STMOPA (2-way)	FEAT_SME_TMOP
1	UTMOPA (2-way)	FEAT_SME_TMOP

SME Int8 to Int32 sparse outer product

The encodings in this section are decoded from SME2 Sparse Outer Product.

31						25	24	23	22	21	20			16	15	14	13	12	11		10	9			6	5		4	3	2	1		0
1	0	0	0	0	0	0	u0	0	1	u1	Zm				1	0	0	K	Zk			Zn				i2			0	0	ZAda		

u0	u1	Instruction Details	Feature
0	0	STMOPA (4-way)	FEAT_SME_TMOP
0	1	SUTMOPA	FEAT_SME_TMOP
1	0	USTMOPA	FEAT_SME_TMOP
1	1	UTMOPA (4-way)	FEAT_SME_TMOP

SME FP Outer Product - 32 bit

SME FP Outer Product - 32 bit

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20																					5	4	3	2	1		0
1	0	0	0	0	0	0	op0	1	0	op1																							op2	0	0			

op0	op1	op2	Instruction details	Feature
0	0	x	SME FP32 outer product	-
0	1	0	FMOPA (widening, 4-way)	FEAT_SME_F8F32
0	1	1	UNALLOCATED	-
1	0	x	SME BF16 widening outer product	-
1	1	x	SME FP16 widening outer product	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME FP32 outer product

The encodings in this section are decoded from SME FP Outer Product - 32 bit.

31–25	24	23–22	21	20–16	15–13	12–10	9–5	4	3–2	1–0
1 0 0 0 0 0 0	0	1 0	0	Zm	Pm	Pn	Zn	S	0 0	ZAda

S	Instruction Details	Feature
0	FMOPA (non-widening)	FEAT_SME
1	FMOPS (non-widening)	FEAT_SME

SME BF16 widening outer product

The encodings in this section are decoded from SME FP Outer Product - 32 bit.

31–25	24	23–22	21	20–16	15–13	12–10	9–5	4	3–2	1–0
1 0 0 0 0 0 0	1	1 0	0	Zm	Pm	Pn	Zn	S	0 0	ZAda

S	Instruction Details	Feature
0	BFMOPA (widening)	FEAT_SME
1	BFMOPS (widening)	FEAT_SME

SME FP16 widening outer product

The encodings in this section are decoded from SME FP Outer Product - 32 bit.

31–25	24	23–22	21	20–16	15–13	12–10	9–5	4	3–2	1–0
1 0 0 0 0 0 0	1	1 0	1	Zm	Pm	Pn	Zn	S	0 0	ZAda

S	Instruction Details	Feature
0	FMOPA (widening, 2-way, FP16 to FP32)	FEAT_SME
1	FMOPS (widening)	FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 Outer Product - Misc

SME2 Outer Product - Misc

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20	5	4	3	2	1	0
1	0	0	0	0	0	0	op0	1	0	op1			op2	1	0	op3	

op0	op1	op2	op3	Instruction details	Feature
0	0	x	x	SME2 32-bit binary outer product	-
0	1	0	0	FMOPA (widening, 2-way, FP8 to FP16)	FEAT_SME_F8F16
0	1	1	0	UNALLOCATED	-
0	1	x	1	UNALLOCATED	-
1	0	x	0	SME2 FP16 non-widening outer product	-
1	1	x	0	SME2 BF16 non-widening outer product	-
1	x	x	1	UNALLOCATED	-

SME2 32-bit binary outer product

The encodings in this section are decoded from SME2 Outer Product - Misc.

31						25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	0	0	Zm		Pm		Pn		Zn		S	1	0	ZAda	

S	Instruction Details	Feature
0	BMOPA	FEAT_SME2
1	BMOPS	FEAT_SME2

SME2 FP16 non-widening outer product

The encodings in this section are decoded from SME2 Outer Product - Misc.

31						25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	0	Zm		Pm		Pn		Zn		S	1	0	0	ZAda

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	Instruction Details	Feature
0	FMOPA (non-widening)	FEAT_SME_F16F16
1	FMOPS (non-widening)	FEAT_SME_F16F16

SME2 BF16 non-widening outer product

The encodings in this section are decoded from SME2 Outer Product - Misc.

31						25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	1	Zm		Pm		Pn		Zn		S	1	0	0	ZAda

S	Instruction Details	Feature
0	BFMOPA (non-widening)	FEAT_SME_B16B16
1	BFMOPS (non-widening)	FEAT_SME_B16B16

SME2 Multi-vector - Memory (Contiguous)

SME2 Multi-vector - Memory (Contiguous)

The encodings in this section are decoded from SME encodings.

31								23	22	20	19	16	15	14	2	1	0
1	0	1	0	0	0	0	0	0	op0				op1			op2	

op0	op1	op2	Instruction details
00x	0	x	SME2 multi-vec contiguous load (scalar plus scalar, two registers)
00x	1	0	SME2 multi-vec contiguous load (scalar plus scalar, four registers)
01x	0	x	SME2 multi-vec contiguous store (scalar plus scalar, two registers)
01x	1	0	SME2 multi-vec contiguous store (scalar plus scalar, four registers)
0x1	1	1	UNALLOCATED
100	0	x	SME2 multi-vec contiguous load (scalar plus immediate, two registers)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
100	1	0	SME2 multi-vec contiguous load (scalar plus immediate, four registers)
110	0	x	SME2 multi-vec contiguous store (scalar plus immediate, two registers)
110	1	0	SME2 multi-vec contiguous store (scalar plus immediate, four registers)
1x1	x	x	UNALLOCATED
xx0	1	1	UNALLOCATED

SME2 multi-vec contiguous load (scalar plus scalar, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31										21	20		16	15	14	13	12		10	9			5	4			1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			0	msz		PNg			Rn				Zt				N

msz	N	Instruction Details	Feature
00	0	LD1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	LDNT1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	LD1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	LDNT1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	LD1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	LDNT1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	LD1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	LDNT1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous load (scalar plus scalar, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31										21	20		16	15	14	13	12		10	9			5	4			2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			1	msz		PNg			Rn				Zt				0	N

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Feature
00	0	LD1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	LDNT1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	LD1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	LDNT1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	LD1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	LDNT1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	LD1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	LDNT1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous store (scalar plus scalar, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31										21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm		0	msz		PNg		Rn		Zt		N

msz	N	Instruction Details	Feature
00	0	ST1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	STNT1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	ST1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	STNT1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	ST1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	STNT1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	ST1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	STNT1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec contiguous store (scalar plus scalar, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31–21	20–16	15	14–13	12–10	9–5	4–2	1	0
1 0 1 0 0 0 0 0 0 0 1	Rm	1	msz	PNg	Rn	Zt	0	N

msz	N	Instruction Details	Feature
00	0	ST1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	STNT1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	ST1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	STNT1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	ST1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	STNT1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	ST1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	STNT1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous load (scalar plus immediate, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31–23	22–20	19–16	15	14–13	12–10	9–5	4–1	0
1 0 1 0 0 0 0 0 0	1 0 0	imm4	0	msz	PNg	Rn	Zt	N

msz	N	Instruction Details	Feature
00	0	LD1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	LDNT1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Feature
01	0	LD1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	LDNT1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	LD1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	LDNT1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	LD1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	LDNT1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous load (scalar plus immediate, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31–23	22–20	19–16	15	14–13	12–10	9–5	4–2	1	0
1 0 1 0 0 0 0 0 0	1 0 0	imm4	1	msz	PNg	Rn	Zt	0	N

msz	N	Instruction Details	Feature
00	0	LD1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	LDNT1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	LD1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	LDNT1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	LD1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	LDNT1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	LD1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	LDNT1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous store (scalar plus immediate, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31								23	22		20	19			16	15	14	13	12			10	9			5	4			1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4				0	msz		PNg				Rn				Zt				N

msz	N	Instruction Details	Feature
00	0	ST1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	STNT1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	ST1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	STNT1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	ST1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	STNT1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	ST1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	STNT1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous store (scalar plus immediate, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31								23	22		20	19			16	15	14	13	12			10	9			5	4			2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4				1	msz		PNg				Rn				Zt				0	N

msz	N	Instruction Details	Feature
00	0	ST1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	STNT1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	ST1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	STNT1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	ST1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Feature
10	1	STNT1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	ST1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	STNT1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 Multi-vector - Memory (Strided)

SME2 Multi-vector - Memory (Strided)

The encodings in this section are decoded from SME encodings.

31								23	22		20	19		16	15	14		3	2	1	0
1	0	1	0	0	0	0	1	0	op0						op1				op2		

op0	op1	op2	Instruction details
00x	0	x	SME2 multi-vec non-contiguous load (scalar plus scalar, two registers)
00x	1	0	SME2 multi-vec non-contiguous load (scalar plus scalar, four registers)
01x	0	x	SME2 multi-vec non-contiguous store (scalar plus scalar, two registers)
01x	1	0	SME2 multi-vec non-contiguous store (scalar plus scalar, four registers)
0x1	1	1	UNALLOCATED
100	0	x	SME2 multi-vec non-contiguous load (scalar plus immediate, two registers)
100	1	0	SME2 multi-vec non-contiguous load (scalar plus immediate, four registers)
110	0	x	SME2 multi-vec non-contiguous store (scalar plus immediate, two registers)
110	1	0	SME2 multi-vec non-contiguous store (scalar plus immediate, four registers)
1x1	x	x	UNALLOCATED
xx0	1	1	UNALLOCATED

SME2 multi-vec non-contiguous load (scalar plus scalar, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31										21	20		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	0	0	Rm			0	msz		PNg			Rn			T	N	Zt		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Feature
00	0	LD1B (scalar plus scalar, strided registers)	FEAT_SME2
00	1	LDNT1B (scalar plus scalar, strided registers)	FEAT_SME2
01	0	LD1H (scalar plus scalar, strided registers)	FEAT_SME2
01	1	LDNT1H (scalar plus scalar, strided registers)	FEAT_SME2
10	0	LD1W (scalar plus scalar, strided registers)	FEAT_SME2
10	1	LDNT1W (scalar plus scalar, strided registers)	FEAT_SME2
11	0	LD1D (scalar plus scalar, strided registers)	FEAT_SME2
11	1	LDNT1D (scalar plus scalar, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous load (scalar plus scalar, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31										21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm		1	msz		PNg		Rn		T	N	0	Zt	

msz	N	Instruction Details	Feature
00	0	LD1B (scalar plus scalar, strided registers)	FEAT_SME2
00	1	LDNT1B (scalar plus scalar, strided registers)	FEAT_SME2
01	0	LD1H (scalar plus scalar, strided registers)	FEAT_SME2
01	1	LDNT1H (scalar plus scalar, strided registers)	FEAT_SME2
10	0	LD1W (scalar plus scalar, strided registers)	FEAT_SME2
10	1	LDNT1W (scalar plus scalar, strided registers)	FEAT_SME2
11	0	LD1D (scalar plus scalar, strided registers)	FEAT_SME2
11	1	LDNT1D (scalar plus scalar, strided registers)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec non-contiguous store (scalar plus scalar, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31										21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	0	1	Rm		0	msz		PNg		Rn		T	N	Zt	

msz	N	Instruction Details	Feature
00	0	ST1B (scalar plus scalar, strided registers)	FEAT_SME2
00	1	STNT1B (scalar plus scalar, strided registers)	FEAT_SME2
01	0	ST1H (scalar plus scalar, strided registers)	FEAT_SME2
01	1	STNT1H (scalar plus scalar, strided registers)	FEAT_SME2
10	0	ST1W (scalar plus scalar, strided registers)	FEAT_SME2
10	1	STNT1W (scalar plus scalar, strided registers)	FEAT_SME2
11	0	ST1D (scalar plus scalar, strided registers)	FEAT_SME2
11	1	STNT1D (scalar plus scalar, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous store (scalar plus scalar, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31										21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm		1	msz		PNg		Rn		T	N	0	Zt	

msz	N	Instruction Details	Feature
00	0	ST1B (scalar plus scalar, strided registers)	FEAT_SME2
00	1	STNT1B (scalar plus scalar, strided registers)	FEAT_SME2
01	0	ST1H (scalar plus scalar, strided registers)	FEAT_SME2
01	1	STNT1H (scalar plus scalar, strided registers)	FEAT_SME2
10	0	ST1W (scalar plus scalar, strided registers)	FEAT_SME2
10	1	STNT1W (scalar plus scalar, strided registers)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Feature
11	0	ST1D (scalar plus scalar, strided registers)	FEAT_SME2
11	1	STNT1D (scalar plus scalar, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous load (scalar plus immediate, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31								23	22		20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4		0	msz		PNg		Rn		T	N	Zt	

msz	N	Instruction Details	Feature
00	0	LD1B (scalar plus immediate, strided registers)	FEAT_SME2
00	1	LDNT1B (scalar plus immediate, strided registers)	FEAT_SME2
01	0	LD1H (scalar plus immediate, strided registers)	FEAT_SME2
01	1	LDNT1H (scalar plus immediate, strided registers)	FEAT_SME2
10	0	LD1W (scalar plus immediate, strided registers)	FEAT_SME2
10	1	LDNT1W (scalar plus immediate, strided registers)	FEAT_SME2
11	0	LD1D (scalar plus immediate, strided registers)	FEAT_SME2
11	1	LDNT1D (scalar plus immediate, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous load (scalar plus immediate, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31								23	22		20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4		1	msz		PNg		Rn		T	N	0	Zt	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Feature
00	0	LD1B (scalar plus immediate, strided registers)	FEAT_SME2
00	1	LDNT1B (scalar plus immediate, strided registers)	FEAT_SME2
01	0	LD1H (scalar plus immediate, strided registers)	FEAT_SME2
01	1	LDNT1H (scalar plus immediate, strided registers)	FEAT_SME2
10	0	LD1W (scalar plus immediate, strided registers)	FEAT_SME2
10	1	LDNT1W (scalar plus immediate, strided registers)	FEAT_SME2
11	0	LD1D (scalar plus immediate, strided registers)	FEAT_SME2
11	1	LDNT1D (scalar plus immediate, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous store (scalar plus immediate, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31								23	22		20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4		0	msz		PNg		Rn		T	N	Zt	

msz	N	Instruction Details	Feature
00	0	ST1B (scalar plus immediate, strided registers)	FEAT_SME2
00	1	STNT1B (scalar plus immediate, strided registers)	FEAT_SME2
01	0	ST1H (scalar plus immediate, strided registers)	FEAT_SME2
01	1	STNT1H (scalar plus immediate, strided registers)	FEAT_SME2
10	0	ST1W (scalar plus immediate, strided registers)	FEAT_SME2
10	1	STNT1W (scalar plus immediate, strided registers)	FEAT_SME2
11	0	ST1D (scalar plus immediate, strided registers)	FEAT_SME2
11	1	STNT1D (scalar plus immediate, strided registers)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec non-contiguous store (scalar plus immediate, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31–23	22–20	19–16	15	14–13	12–10	9–5	4	3	2	1–0
1 0 1 0 0 0 0 1 0	1 1 0	imm4	1	msz	PNg	Rn	T	N	0	Zt

msz	N	Instruction Details	Feature
00	0	ST1B (scalar plus immediate, strided registers)	FEAT_SME2
00	1	STNT1B (scalar plus immediate, strided registers)	FEAT_SME2
01	0	ST1H (scalar plus immediate, strided registers)	FEAT_SME2
01	1	STNT1H (scalar plus immediate, strided registers)	FEAT_SME2
10	0	ST1W (scalar plus immediate, strided registers)	FEAT_SME2
10	1	STNT1W (scalar plus immediate, strided registers)	FEAT_SME2
11	0	ST1D (scalar plus immediate, strided registers)	FEAT_SME2
11	1	STNT1D (scalar plus immediate, strided registers)	FEAT_SME2

SME Integer Outer Product - 32 bit

SME Integer Outer Product - 32 bit

The encodings in this section are decoded from SME encodings.

31–25	24	23–22	21	20–4	3	2	1–0
1 0 1 0 0 0 0		1 0	op0		op1	0	

op0	op1	Instruction details
0	1	SME2 Int16 two-way outer product
1	1	UNALLOCATED
x	0	SME Int8 outer product

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 Int16 two-way outer product

The encodings in this section are decoded from SME Integer Outer Product - 32 bit.

31						25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	u0	1	0	0	Zm		Pm		Pn		Zn		S	1	0	ZAda	

u0	S	Instruction Details	Feature
0	0	SMOPA (2-way)	FEAT_SME2
0	1	SMOPS (2-way)	FEAT_SME2
1	0	UMOPA (2-way)	FEAT_SME2
1	1	UMOPS (2-way)	FEAT_SME2

SME Int8 outer product

The encodings in this section are decoded from SME Integer Outer Product - 32 bit.

31						25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	u0	1	0	u1	Zm		Pm		Pn		Zn		S	0	0	ZAda	

u0	u1	S	Instruction Details	Feature
0	0	0	SMOPA (4-way)	FEAT_SME
0	0	1	SMOPS (4-way)	FEAT_SME
0	1	0	SUMOPA (4-way)	FEAT_SME
0	1	1	SUMOPS	FEAT_SME
1	0	0	USMOPA (4-way)	FEAT_SME
1	0	1	USMOPS	FEAT_SME
1	1	0	UMOPA (4-way)	FEAT_SME
1	1	1	UMOPS (4-way)	FEAT_SME

SME2 Quarter Tile Outer Product - 64-bit

SME2 Quarter Tile Outer Product - 64-bit

The encodings in this section are decoded from SME encodings.

31	30	29	28			25	24	23	22	21	20	17	16						10	9	6	5	4	3	2	0
1	0	op0	0	0	0	0	op1	1	1	op2			0	0	0	0	0	0	0			0		1		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
0	0	0	SME2 FP64 non-widening quarter tile outer product
0	0	1	UNALLOCATED
0	1	x	UNALLOCATED
1	x	x	SME2 Int16 to Int64 quarter tile outer product

SME2 FP64 non-widening quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 64-bit.

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	0	0	0	0	1	1	0	M	Zm		0	0	N	Zn		0	S	1	ZAda	

M	N	S	Instruction Details	Feature
0	0	0	FMOP4A (non-widening) — Double-precision, single vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
0	0	1	FMOP4S (non-widening) — Double-precision, single vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
0	1	0	FMOP4A (non-widening) — Double-precision, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
0	1	1	FMOP4S (non-widening) — Double-precision, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
1	0	0	FMOP4A (non-widening) — Double-precision, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
1	0	1	FMOP4S (non-widening) — Double-precision, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
1	1	0	FMOP4A (non-widening) — Double-precision, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
1	1	1	FMOP4S (non-widening) — Double-precision, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64

SME2 Int16 to Int64 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 64-bit.

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	1	0	0	u0	1	1	u1	M	Zm		0	0	N	Zn		0	S	1	ZAda	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	u1	M	N	S	Instruction Details	Feature
0	0	0	0	0	SMOP4A (4-way) — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	0	0	1	SMOP4S (4-way) — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	0	1	0	SMOP4A (4-way) — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	0	1	1	SMOP4S (4-way) — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	1	0	0	SMOP4A (4-way) — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	1	0	1	SMOP4S (4-way) — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	1	1	0	SMOP4A (4-way) — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	1	1	1	SMOP4S (4-way) — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	0	0	0	SUMOP4A — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	0	0	1	SUMOP4S — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	0	1	0	SUMOP4A — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	0	1	1	SUMOP4S — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	1	0	0	SUMOP4A — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	1	0	1	SUMOP4S — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	1	1	0	SUMOP4A — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	1	1	1	SUMOP4S — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	0	0	0	USMOP4A — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	0	0	1	USMOP4S — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	0	1	0	USMOP4A — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	0	1	1	USMOP4S — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	1	0	0	USMOP4A — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	1	0	1	USMOP4S — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	1	1	0	USMOP4A — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	1	1	1	USMOP4S — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	0	0	0	UMOP4A (4-way) — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	0	0	1	UMOP4S (4-way) — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	0	1	0	UMOP4A (4-way) — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	u1	M	N	S	Instruction Details	Feature
1	1	0	1	1	UMOP4S (4-way) — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	1	0	0	UMOP4A (4-way) — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	1	0	1	UMOP4S (4-way) — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	1	1	0	UMOP4A (4-way) — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	1	1	1	UMOP4S (4-way) — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64

SME Outer Product - 64 bit

SME Outer Product - 64 bit

The encodings in this section are decoded from SME encodings.

31	30	29	28			25	24	23	22	21	20 … 4	3	2 … 0
1	0	op0	0	0	0	0	op1	1	1	op2		0	

op0	op1	op2	Instruction details
0	0	0	SME FP64 outer product
0	0	1	UNALLOCATED
0	1	x	UNALLOCATED
1	x	x	SME Int16 outer product

SME FP64 outer product

The encodings in this section are decoded from SME Outer Product - 64 bit.

31	30	29	28			25	24	23	22	21	20 … 16	15 … 13	12 … 10	9 … 5	4	3	2 … 0
1	0	0	0	0	0	0	0	1	1	0	Zm	Pm	Pn	Zn	S	0	ZAda

S	Instruction Details	Feature
0	FMOPA (non-widening)	FEAT_SME_F64F64
1	FMOPS (non-widening)	FEAT_SME_F64F64

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME Int16 outer product

The encodings in this section are decoded from SME Outer Product - 64 bit.

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	u0	1	1	u1	Zm		Pm		Pn		Zn		S	0	ZAda	

u0	u1	S	Instruction Details	Feature
0	0	0	SMOPA (4-way)	FEAT_SME_I16I64
0	0	1	SMOPS (4-way)	FEAT_SME_I16I64
0	1	0	SUMOPA (4-way)	FEAT_SME_I16I64
0	1	1	SUMOPS	FEAT_SME_I16I64
1	0	0	USMOPA (4-way)	FEAT_SME_I16I64
1	0	1	USMOPS	FEAT_SME_I16I64
1	1	0	UMOPA (4-way)	FEAT_SME_I16I64
1	1	1	UMOPS (4-way)	FEAT_SME_I16I64

SME zero array

SME zero array

The encodings in this section are decoded from SME encodings.

31												18	17	8	7	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	op0		

op0	Instruction details	Feature
0000000000	ZERO (tiles)	FEAT_SME
!= 0000000000	UNALLOCATED	-

SME2 Multiple Zero

SME2 Multiple Zero

The encodings in this section are decoded from SME encodings.

31												18	17	13	12	3	2	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1		op0			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0000000000	SME multiple vectors zero array
!= 0000000000	UNALLOCATED

SME multiple vectors zero array

The encodings in this section are decoded from SME2 Multiple Zero.

31													18	17		15	14	13	12									3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	opc			Rv		0	0	0	0	0	0	0	0	0	0	opc2	

opc	opc2	Instruction Details	Feature
x1x	1xx	UNALLOCATED	-
000	xxx	ZERO (single-vector) — Two ZA single-vectors	FEAT_SME2p1
001	xxx	ZERO (double-vector) — One ZA double-vector	FEAT_SME2p1
010	0xx	ZERO (double-vector) — Two ZA double-vectors	FEAT_SME2p1
011	0xx	ZERO (double-vector) — Four ZA double-vectors	FEAT_SME2p1
100	xxx	ZERO (single-vector) — Four ZA single-vectors	FEAT_SME2p1
101	0xx	ZERO (quad-vector) — One ZA quad-vector	FEAT_SME2p1
101	1xx	UNALLOCATED	-
11x	01x	UNALLOCATED	-
110	00x	ZERO (quad-vector) — Two ZA quad-vectors	FEAT_SME2p1
111	00x	ZERO (quad-vector) — Four ZA quad-vectors	FEAT_SME2p1

SME2 zero lookup table

The encodings in this section are decoded from SME encodings.

31													18	17		4	3		0
1	1	0	0	0	0	0	0	0	1	0	0	1	0	op0			opc		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	opc	Instruction Details	Feature
00000000000000	0001	ZERO (table)	FEAT_SME2
00000000000000	!= 0001	UNALLOCATED	-
!= 00000000000000	xxxx	UNALLOCATED	-

SME2 Move Lookup Table

SME2 Move Lookup Table

The encodings in this section are decoded from SME encodings.

31													18	17	16	15	14	13	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	op0	op1		op2		

op0	op1	op2	Instruction details
0	00	x	SME2 move from lookup table
0	10	x	UNALLOCATED
1	00	x	SME2 move into lookup table
1	10	0	SME2 move vector to lookup table
1	10	1	UNALLOCATED
x	x1	x	UNALLOCATED

SME2 move from lookup table

The encodings in this section are decoded from SME2 Move Lookup Table.

31													18	17	16	15	14	12	11	5	4	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	imm3		opc		Rt	

opc	Instruction Details	Feature
0011111	MOVT (table to scalar)	FEAT_SME2
!= 0011111	UNALLOCATED	-

SME2 move into lookup table

The encodings in this section are decoded from SME2 Move Lookup Table.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31													18	17	16	15	14	12	11	5	4	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	imm3		opc		Rt	

opc	Instruction Details	Feature
0011111	MOVT (scalar to table)	FEAT_SME2
!= 0011111	UNALLOCATED	-

SME2 move vector to lookup table

The encodings in this section are decoded from SME2 Move Lookup Table.

31													18	17	16	15	14	13	12	11	5	4	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0	0	off2		opc		Zt	

opc	Instruction Details	Feature
0011111	MOVT (vector to table)	FEAT_SME_LUTv2
!= 0011111	UNALLOCATED	-

SME2 Expand Lookup Table (Non-contiguous)

SME2 Expand Lookup Table (Non-contiguous)

The encodings in this section are decoded from SME encodings.

31												19	18	16	15	14	13	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	op0		op1				op2		op3			

op0	op1	op2	op3	Instruction details
011	00	0	00	SME2 lookup table two source registers expand to four non-contiguous destination registers
011	00	1	00	UNALLOCATED
xxx	10	x	00	SME2 lookup table expand four non-contiguous registers
xxx	x0	x	01	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details
xxx	x1	x	0x	SME2 lookup table expand two non-contiguous registers
xxx	xx	x	1x	UNALLOCATED
!=011	00	x	00	UNALLOCATED

SME2 lookup table two source registers expand to four non-contiguous destination registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Non-contiguous).

31												19	18		16	15	14	13	12	11	10	9		6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	size		opc		Zn			0	D	0	0	Zd	

opc	Instruction Details	Feature
00	LUTI4 (four registers, 8-bit)	FEAT_SME2p1 && FEAT_SME_LUTv2
!=00	UNALLOCATED	-

SME2 lookup table expand four non-contiguous registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Non-contiguous).

31												19	18		16	15	14	13	12	11	10	9		5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1		opc		1	0	size		opc2		Zn			D	0	0	Zd	

opc	opc2	Instruction Details	Feature
xxx	!=00	UNALLOCATED	-
00x	00	UNALLOCATED	-
01x	00	LUTI4 (four registers, 16-bit and 32-bit)	FEAT_SME2p1
1xx	00	LUTI2 (four registers)	FEAT_SME2p1

SME2 lookup table expand two non-contiguous registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Non-contiguous).

31												19	18		15	14	13	12	11	10	9		5	4	3	2		0
1	1	0	0	0	0	0	0	1	0	0	1	1		opc		1		size		opc2		Zn		D	0		Zd	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	Instruction Details	Feature
xxxx	!= 00	UNALLOCATED	-
00xx	00	UNALLOCATED	-
01xx	00	LUTI4 (two registers)	FEAT_SME2p1
1xxx	00	LUTI2 (two registers)	FEAT_SME2p1

SME2 Expand Lookup Table (Contiguous)

SME2 Expand Lookup Table (Contiguous)

The encodings in this section are decoded from SME encodings.

31	23	22	21	19	18	14	13	6	5	4	2	1	0
1 1 0 0 0 0 0 0 1		op0	0 0 1		op1				op2			op3	

op0	op1	op2	op3	Instruction details
0	00x00	x	00	UNALLOCATED
0	01000	x	00	UNALLOCATED
0	01100	0	00	SME2 lookup table two source registers expand to four contiguous destination registers
0	01100	1	00	UNALLOCATED
0	1xx00	x	00	UNALLOCATED
0	xxx10	x	00	SME2 lookup table expand four contiguous registers
0	xxxx0	x	10	UNALLOCATED
0	xxxx1	x	x0	SME2 lookup table expand two contiguous registers
0	xxxxx	x	x1	UNALLOCATED
1	xxxxx	x	xx	SME2 lookup table expand one register

SME2 lookup table two source registers expand to four contiguous destination registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Contiguous).

31	23	22	21	19	18	14	13	12	11	10	9	6	5	4	2	1	0
1 1 0 0 0 0 0 0 1		0	0 0 1		0 1 1 0 0		size		opc		Zn		0	Zd		0	0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
00	LUTI4 (four registers, 8-bit)	FEAT_SME_LUTv2
!= 00	UNALLOCATED	-

SME2 lookup table expand four contiguous registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Contiguous).

31								23	22	21		19	18		16	15	14	13		12	11		10	9		5	4		2	1		0
1	1	0	0	0	0	0	0	1	0	0	0	1	opc			1	0	size			opc2			Zn			Zd			0		0

opc	opc2	Instruction Details	Feature
xxx	!= 00	UNALLOCATED	-
00x	00	UNALLOCATED	-
01x	00	LUTI4 (four registers, 16-bit and 32-bit)	FEAT_SME2
1xx	00	LUTI2 (four registers)	FEAT_SME2

SME2 lookup table expand two contiguous registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Contiguous).

31								23	22	21		19	18		15	14	13		12	11		10	9		5	4		1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	opc			1	size			opc2			Zn			Zd			0

opc	opc2	Instruction Details	Feature
xxxx	!= 00	UNALLOCATED	-
00xx	00	UNALLOCATED	-
01xx	00	LUTI4 (two registers)	FEAT_SME2
1xxx	00	LUTI2 (two registers)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 lookup table expand one register

The encodings in this section are decoded from SME2 Expand Lookup Table (Contiguous).

31–23	22	21–19	18–14	13–12	11–10	9–5	4–0
1 1 0 0 0 0 0 0 1	1	0 0 1	opc	size	opc2	Zn	Zd

opc	opc2	Instruction Details	Feature
xxxxx	!=00	UNALLOCATED	-
00xxx	00	UNALLOCATED	-
01xxx	00	LUTI4 (single)	FEAT_SME2
1xxxx	00	LUTI2 (single)	FEAT_SME2

SME Move into Array

SME Move into Array

The encodings in this section are decoded from SME encodings.

31–24	23–22	21–19	18	17	16–15	14–13	12–10	9–7	6–5	4	3	2–0
1 1 0 0 0 0 0 0	op0	0 0 0	op1	0	op2		op3		op4	0	op5	

op0	op1	op2	op3	op4	op5	Instruction details	Feature
00	1	00	010	x0	0	MOVA (vector to array, two registers)	FEAT_SME2
00	1	00	011	00	0	MOVA (vector to array, four registers)	FEAT_SME2
00	1	00	011	10	0	UNALLOCATED	-
00	1	01	01x	x0	0	UNALLOCATED	-
xx	0	xx	xxx	xx	x	SME move vector to array	-
xx	1	0x	000	x0	0	SME2 move vector to tile, two registers	-
xx	1	0x	001	00	0	SME2 move vector to tile, four registers	-
xx	1	0x	001	10	0	UNALLOCATED	-
xx	1	0x	0xx	x0	1	UNALLOCATED	-
xx	1	0x	0xx	x1	x	UNALLOCATED	-
xx	1	0x	1xx	xx	x	UNALLOCATED	-
xx	1	1x	xxx	xx	x	UNALLOCATED	-
!=00	1	0x	01x	x0	0	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME move vector to array

The encodings in this section are decoded from SME Move into Array.

31							24	23	22	21		19	18	17	16	15	14	13	12		10	9		5	4	3		0
1	1	0	0	0	0	0	0	size		0	0	0	0	0	Q	V	Rs		Pg			Zn			0	opc		

size	Q	Instruction Details	Feature
00	0	MOVA (vector to tile, single) — 8-bit	FEAT_SME
01	0	MOVA (vector to tile, single) — 16-bit	FEAT_SME
10	0	MOVA (vector to tile, single) — 32-bit	FEAT_SME
11	0	MOVA (vector to tile, single) — 64-bit	FEAT_SME
11	1	MOVA (vector to tile, single) — 128-bit	FEAT_SME
!= 11	1	UNALLOCATED	-

SME2 move vector to tile, two registers

The encodings in this section are decoded from SME Move into Array.

31							24	23	22	21		19	18	17		16	15	14	13	12			10	9		6	5		4	3	2		0
1	1	0	0	0	0	0	0	size		0	0	0	1	0		0	V	Rs		0	0		0	Zn			0		0	0	opc		

size	Instruction Details	Feature
00	MOVA (vector to tile, two registers) — 8-bit	FEAT_SME2
01	MOVA (vector to tile, two registers) — 16-bit	FEAT_SME2
10	MOVA (vector to tile, two registers) — 32-bit	FEAT_SME2
11	MOVA (vector to tile, two registers) — 64-bit	FEAT_SME2

SME2 move vector to tile, four registers

The encodings in this section are decoded from SME Move into Array.

31							24	23	22	21		19	18	17		16	15	14	13	12			10	9		7	6		5	4	3	2		0
1	1	0	0	0	0	0	0	size		0	0	0	1	0		0	V	Rs		0	0		1	Zn			0		0	0	0	opc		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Feature
00	0xx	MOVA (vector to tile, four registers) — 8-bit	FEAT_SME2
01	0xx	MOVA (vector to tile, four registers) — 16-bit	FEAT_SME2
10	0xx	MOVA (vector to tile, four registers) — 32-bit	FEAT_SME2
11	xxx	MOVA (vector to tile, four registers) — 64-bit	FEAT_SME2
!= 11	1xx	UNALLOCATED	-

SME Move from Array

SME Move from Array

The encodings in this section are decoded from SME encodings.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7					2	1	0
1	1	0	0	0	0	0	0	op0		0	0	0		op1	1	op2				op3				op4								op5	

op0	op1	op2	op3	op4	op5	Instruction details	Feature
00	1	00	010	00	x0	MOVA (array to vector, two registers)	FEAT_SME2
00	1	00	010	10	x0	MOVAZ (array to vector, two registers)	FEAT_SME2p1
00	1	00	011	00	00	MOVA (array to vector, four registers)	FEAT_SME2
00	1	00	011	10	00	MOVAZ (array to vector, four registers)	FEAT_SME2p1
00	1	00	011	x0	10	UNALLOCATED	-
00	1	01	01x	x0	x0	UNALLOCATED	-
xx	0	xx	000	1x	xx	SME zeroing move array to vector	-
xx	0	xx	xxx	0x	xx	SME move array to vector	-
xx	0	xx	!= 000	1x	xx	UNALLOCATED	-
xx	1	0x	000	00	x0	SME2 move tile to vector, two registers	-
xx	1	0x	000	10	x0	SME2 zeroing move tile to vector, two registers	-
xx	1	0x	001	00	00	SME2 move tile to vector, four registers	-
xx	1	0x	001	10	00	SME2 zeroing move tile to vector, four registers	-
xx	1	0x	001	x0	10	UNALLOCATED	-
xx	1	0x	0xx	x0	x1	UNALLOCATED	-
xx	1	0x	0xx	x1	xx	UNALLOCATED	-
xx	1	0x	1xx	xx	xx	UNALLOCATED	-
xx	1	1x	xxx	xx	xx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	op5	Instruction details	Feature
!=00	1	0x	01x	x0	x0	UNALLOCATED	-

SME zeroing move array to vector

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			5	4			0
1	1	0	0	0	0	0	0	size		0	0	0		0	1	Q	V	Rs		0	0	0		1	opc				Zd			

size	Q	Instruction Details	Feature
00	0	MOVAZ (tile to vector, single) — 8-bit	FEAT_SME2p1
01	0	MOVAZ (tile to vector, single) — 16-bit	FEAT_SME2p1
10	0	MOVAZ (tile to vector, single) — 32-bit	FEAT_SME2p1
11	0	MOVAZ (tile to vector, single) — 64-bit	FEAT_SME2p1
11	1	MOVAZ (tile to vector, single) — 128-bit	FEAT_SME2p1
!=11	1	UNALLOCATED	-

SME move array to vector

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			5	4			0
1	1	0	0	0	0	0	0	size		0	0	0		0	1	Q	V	Rs		Pg				0	opc				Zd			

size	Q	Instruction Details	Feature
00	0	MOVA (tile to vector, single) — 8-bit	FEAT_SME
01	0	MOVA (tile to vector, single) — 16-bit	FEAT_SME
10	0	MOVA (tile to vector, single) — 32-bit	FEAT_SME
11	0	MOVA (tile to vector, single) — 64-bit	FEAT_SME
11	1	MOVA (tile to vector, single) — 128-bit	FEAT_SME
!=11	1	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 move tile to vector, two registers

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	0	size		0	0	0		1	1	0	V	Rs		0	0	0		0	0	opc				Zd			0

size	Instruction Details	Feature
00	MOVA (tile to vector, two registers) — 8-bit	FEAT_SME2
01	MOVA (tile to vector, two registers) — 16-bit	FEAT_SME2
10	MOVA (tile to vector, two registers) — 32-bit	FEAT_SME2
11	MOVA (tile to vector, two registers) — 64-bit	FEAT_SME2

SME2 zeroing move tile to vector, two registers

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	0	size		0	0	0		1	1	0	V	Rs		0	0	0		1	0	opc				Zd			0

size	Instruction Details	Feature
00	MOVAZ (tile to vector, two registers) — 8-bit	FEAT_SME2p1
01	MOVAZ (tile to vector, two registers) — 16-bit	FEAT_SME2p1
10	MOVAZ (tile to vector, two registers) — 32-bit	FEAT_SME2p1
11	MOVAZ (tile to vector, two registers) — 64-bit	FEAT_SME2p1

SME2 move tile to vector, four registers

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4			2	1	0
1	1	0	0	0	0	0	0	size		0	0	0		1	1	0	V	Rs		0	0	1		0	0	opc				Zd			0	0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Feature
00	0xx	MOVA (tile to vector, four registers) — 8-bit	FEAT_SME2
01	0xx	MOVA (tile to vector, four registers) — 16-bit	FEAT_SME2
10	0xx	MOVA (tile to vector, four registers) — 32-bit	FEAT_SME2
11	xxx	MOVA (tile to vector, four registers) — 64-bit	FEAT_SME2
!= 11	1xx	UNALLOCATED	-

SME2 zeroing move tile to vector, four registers

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	0	size		0	0	0	1	1	0	V	Rs		0	0	1	1	0	opc			Zd			0	0

size	opc	Instruction Details	Feature
00	0xx	MOVAZ (tile to vector, four registers) — 8-bit	FEAT_SME2p1
01	0xx	MOVAZ (tile to vector, four registers) — 16-bit	FEAT_SME2p1
10	0xx	MOVAZ (tile to vector, four registers) — 32-bit	FEAT_SME2p1
11	xxx	MOVAZ (tile to vector, four registers) — 64-bit	FEAT_SME2p1
!= 11	1xx	UNALLOCATED	-

SME Add Vector to Array

SME Add Vector to Array

The encodings in this section are decoded from SME encodings.

31							24	23	22	21		19	18	17	16		5	4	3	2		0
1	1	0	0	0	0	0	0	op0		0	1	0	op1					op2	0			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
0	xx	x	UNALLOCATED
1	00	0	SME add vector to array
1	00	1	UNALLOCATED
1	!=00	x	UNALLOCATED

SME add vector to array

The encodings in this section are decoded from SME Add Vector to Array.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4	3	2		0
1	1	0	0	0	0	0	0	1	op	0	1	0	0	0	V	Pm			Pn			Zn			0	0	opc2		

op	V	opc2	Instruction Details	Feature
0	x	1xx	UNALLOCATED	-
0	0	0xx	ADDHA — 32-bit	FEAT_SME
0	1	0xx	ADDVA — 32-bit	FEAT_SME
1	0	xxx	ADDHA — 64-bit	FEAT_SME_I16I64
1	1	xxx	ADDVA — 64-bit	FEAT_SME_I16I64

SME2 Multi-vector - Multiple and Single Array Vectors (Two registers)

SME2 Multi-vector - Multiple and Single Array Vectors (Two registers)

The encodings in this section are decoded from SME encodings.

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4			2	1	0
1	1	0	0	0	0	0	1	0	op0	1	0					0			op1							op2				op3	

op0	op1	op2	op3	Instruction details	Feature
0	000	000	1	FMLALL (multiple and single vector) — Two ZA quad-vectors	FEAT_SME_F8F32
0	000	!=000	1	UNALLOCATED	-
0	010	xxx	x	SME2 single-multi long FMA two sources	-
0	011	xxx	x	SME2 multiple and single vector long FMA one source	-
0	100	xxx	x	SME2 single-multi FP dot product two registers	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
0	101	x1x	x	SME2 single-multi mixed dot product two registers	-
1	000	xxx	1	UNALLOCATED	-
1	010	xxx	x	SME2 single-multi long MLA two sources	-
1	011	xxx	x	SME2 multiple and single vector long MLA one source	-
1	100	xxx	x	UNALLOCATED	-
1	101	x1x	x	SME2 single-multi two-way dot product two registers	-
x	000	xxx	0	SME2 single-multi long long MLA two sources	-
x	001	xxx	x	SME2 multiple and single vector long long FMA one source	-
x	101	x0x	x	SME2 single-multi four-way dot product two registers	-
x	110	0xx	x	SME2 single-multi ternary FP two registers	-
x	110	1xx	x	SME2 single-multi ternary int two registers	-
x	111	0xx	x	SME2 single-multi ternary FP16 two registers	-
x	111	1xx	x	UNALLOCATED	-

SME2 single-multi long FMA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		0	1	0	Zn				op	S	o2	off2	

op	S	o2	Instruction Details	Feature
0	0	0	FMLAL (multiple and single vector, FP16 to FP32)	FEAT_SME2
0	0	1	FMLAL (multiple and single vector, FP8 to FP16)	FEAT_SME_F8F16
0	1	0	FMLSL (multiple and single vector)	FEAT_SME2
0	1	1	UNALLOCATED	-
1	x	1	UNALLOCATED	-
1	0	0	BFMLAL (multiple and single vector)	FEAT_SME2
1	1	0	BFMLSL (multiple and single vector)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple and single vector long FMA one source

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		0	1	1	Zn				op	S	off3			

op	S	Instruction Details	Feature
0	0	FMLAL (multiple and single vector, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple and single vector)	FEAT_SME2
1	0	BFMLAL (multiple and single vector)	FEAT_SME2
1	1	BFMLSL (multiple and single vector)	FEAT_SME2

SME2 single-multi FP dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4		3	2			0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		1	0	0	Zn				opc			off3			

opc	Instruction Details	Feature
00	FDOT (2-way, multiple and single vector, FP16 to FP32)	FEAT_SME2
01	FDOT (2-way, multiple and single vector, FP8 to FP16)	FEAT_SME_F8F16
10	BFDOT (multiple and single vector)	FEAT_SME2
11	FDOT (4-way, multiple and single vector)	FEAT_SME_F8F32

SME2 single-multi mixed dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		1	0	1	Zn				U	1	off3			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Feature
0	USDOT (4-way, multiple and single vector)	FEAT_SME2
1	SUDOT (4-way, multiple and single vector)	FEAT_SME2

SME2 single-multi long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm				0	Rv		0	1	0	Zn				U	S	op	off2	

U	S	op	Instruction Details	Feature
x	x	1	UNALLOCATED	-
0	0	0	SMLAL (multiple and single vector)	FEAT_SME2
0	1	0	SMLSL (multiple and single vector)	FEAT_SME2
1	0	0	UMLAL (multiple and single vector)	FEAT_SME2
1	1	0	UMLSL (multiple and single vector)	FEAT_SME2

SME2 multiple and single vector long MLA one source

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	1	0	Zm				0	Rv		0	1	1	Zn				U	S	off3		

U	S	Instruction Details	Feature
0	0	SMLAL (multiple and single vector)	FEAT_SME2
0	1	SMLSL (multiple and single vector)	FEAT_SME2
1	0	UMLAL (multiple and single vector)	FEAT_SME2
1	1	UMLSL (multiple and single vector)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi two-way dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2–0
11000001 0	1	10	Zm	0	Rv	101	Zn	U	1	off3

U	Instruction Details	Feature
0	SDOT (2-way, multiple and single vector)	FEAT_SME2
1	UDOT (2-way, multiple and single vector)	FEAT_SME2

SME2 single-multi long long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2	1	0
11000001 0	sz	10	Zm	0	Rv	000	Zn	U	S	op	0	o1

sz	U	S	op	Instruction Details	Feature
x	0	0	0	SMLALL (multiple and single vector)	FEAT_SME2
x	0	1	0	SMLSLL (multiple and single vector)	FEAT_SME2
x	1	0	0	UMLALL (multiple and single vector)	FEAT_SME2
x	1	1	0	UMLSLL (multiple and single vector)	FEAT_SME2
0	x	1	1	UNALLOCATED	-
0	0	0	1	USMLALL (multiple and single vector)	FEAT_SME2
0	1	0	1	SUMLALL (multiple and single vector)	FEAT_SME2
1	x	x	1	UNALLOCATED	-

SME2 multiple and single vector long long FMA one source

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2	1–0
11000001 0	sz	10	Zm	0	Rv	001	Zn	U	S	op	off2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	U	S	op	Instruction Details	Feature
x	0	0	0	SMLALL (multiple and single vector)	FEAT_SME2
x	0	1	0	SMLSLL (multiple and single vector)	FEAT_SME2
x	1	0	0	UMLALL (multiple and single vector)	FEAT_SME2
x	1	1	0	UMLSLL (multiple and single vector)	FEAT_SME2
0	0	0	1	USMLALL (multiple and single vector)	FEAT_SME2
0	0	1	1	UNALLOCATED	-
0	1	x	1	UNALLOCATED	-
1	x	x	1	UNALLOCATED	-

SME2 single-multi four-way dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		1	0	1	Zn				U	0	off3			

U	Instruction Details	Feature
0	SDOT (4-way, multiple and single vector)	FEAT_SME2
1	UDOT (4-way, multiple and single vector)	FEAT_SME2

SME2 single-multi ternary FP two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		1	1	0	Zn				0	S	off3			

S	Instruction Details	Feature
0	FMLA (multiple and single vector)	FEAT_SME2
1	FMLS (multiple and single vector)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi ternary int two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2–0
1 1 0 0 0 0 0 1 0	sz	1 0	Zm	0	Rv	1 1 0	Zn	1	S	off3

S	Instruction Details	Feature
0	ADD (to array, multiple and single vector)	FEAT_SME2
1	SUB (to array, multiple and single vector)	FEAT_SME2

SME2 single-multi ternary FP16 two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2–0
1 1 0 0 0 0 0 1 0	sz	1 0	Zm	0	Rv	1 1 1	Zn	0	S	off3

sz	S	Instruction Details	Feature
0	0	FMLA (multiple and single vector)	FEAT_SME_F16F16
0	1	FMLS (multiple and single vector)	FEAT_SME_F16F16
1	0	BFMLA (multiple and single vector)	FEAT_SME_B16B16
1	1	BFMLS (multiple and single vector)	FEAT_SME_B16B16

SME2 Multi-vector - Multiple and Single Array Vectors (Four registers)

SME2 Multi-vector - Multiple and Single Array Vectors (Four registers)

The encodings in this section are decoded from SME encodings.

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4–2	1	0
1 1 0 0 0 0 0 1 0	op0	1 1		0		op1		op2	op3	

op0	op1	op2	op3	Instruction details	Feature
0	000	000	1	FMLALL (multiple and single vector) — Four ZA quad-vectors	FEAT_SME_F8F32

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
0	000	!=000	1	UNALLOCATED	-
0	001	000	x	FMLALL (multiple and single vector) — One ZA quad-vector	FEAT_SME_F8F32
0	001	001	x	UNALLOCATED	-
0	010	xxx	x	SME2 single-multi long FMA four sources	-
0	011	00x	x	FMLAL (multiple and single vector, FP8 to FP16) — One ZA double-vector	FEAT_SME_F8F16
0	0x1	01x	x	UNALLOCATED	-
0	0x1	1xx	x	UNALLOCATED	-
0	100	xxx	x	SME2 single-multi FP dot product four registers	-
0	101	x1x	x	SME2 single-multi mixed dot product four registers	-
1	000	xxx	1	UNALLOCATED	-
1	010	xxx	x	SME2 single-multi long MLA four sources	-
1	0x1	xxx	x	UNALLOCATED	-
1	100	xxx	x	UNALLOCATED	-
1	101	x1x	x	SME2 single-multi two-way dot product four registers	-
x	000	xxx	0	SME2 single-multi long long MLA four sources	-
x	101	x0x	x	SME2 single-multi four-way dot product four registers	-
x	110	0xx	x	SME2 single-multi ternary FP four registers	-
x	110	1xx	x	SME2 single-multi ternary int four registers	-
x	111	0xx	x	SME2 single-multi ternary FP16 four registers	-
x	111	1xx	x	UNALLOCATED	-

SME2 single-multi long FMA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	1	0	Zn				op	S	o2	off2	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	o2	Instruction Details	Feature
0	0	0	FMLAL (multiple and single vector, FP16 to FP32)	FEAT_SME2
0	0	1	FMLAL (multiple and single vector, FP8 to FP16)	FEAT_SME_F8F16
0	1	0	FMLSL (multiple and single vector)	FEAT_SME2
0	1	1	UNALLOCATED	-
1	x	1	UNALLOCATED	-
1	0	0	BFMLAL (multiple and single vector)	FEAT_SME2
1	1	0	BFMLSL (multiple and single vector)	FEAT_SME2

SME2 single-multi FP dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4		3	2			0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	0	0	Zn				opc			off3			

opc	Instruction Details	Feature
00	FDOT (2-way, multiple and single vector, FP16 to FP32)	FEAT_SME2
01	FDOT (2-way, multiple and single vector, FP8 to FP16)	FEAT_SME_F8F16
10	BFDOT (multiple and single vector)	FEAT_SME2
11	FDOT (4-way, multiple and single vector)	FEAT_SME_F8F32

SME2 single-multi mixed dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	0	1	Zn				U	1	off3			

U	Instruction Details	Feature
0	USDOT (4-way, multiple and single vector)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Feature
1	SUDOT (4-way, multiple and single vector)	FEAT_SME2

SME2 single-multi long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19	16	15	14	13	12		10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm		0	Rv		0	1	0	Zn		U	S	op	off2	

U	S	op	Instruction Details	Feature
x	x	1	UNALLOCATED	-
0	0	0	SMLAL (multiple and single vector)	FEAT_SME2
0	1	0	SMLSL (multiple and single vector)	FEAT_SME2
1	0	0	UMLAL (multiple and single vector)	FEAT_SME2
1	1	0	UMLSL (multiple and single vector)	FEAT_SME2

SME2 single-multi two-way dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19	16	15	14	13	12		10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm		0	Rv		1	0	1	Zn		U	1	off3	

U	Instruction Details	Feature
0	SDOT (2-way, multiple and single vector)	FEAT_SME2
1	UDOT (2-way, multiple and single vector)	FEAT_SME2

SME2 single-multi long long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19	16	15	14	13	12		10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm		0	Rv		0	0	0	Zn		U	S	op	0	o1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	U	S	op	Instruction Details	Feature
x	0	0	0	SMLALL (multiple and single vector)	FEAT_SME2
x	0	1	0	SMLSLL (multiple and single vector)	FEAT_SME2
x	1	0	0	UMLALL (multiple and single vector)	FEAT_SME2
x	1	1	0	UMLSLL (multiple and single vector)	FEAT_SME2
0	x	1	1	UNALLOCATED	-
0	0	0	1	USMLALL (multiple and single vector)	FEAT_SME2
0	1	0	1	SUMLALL (multiple and single vector)	FEAT_SME2
1	x	x	1	UNALLOCATED	-

SME2 single-multi four-way dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		1	0	1	Zn				U	0	off3		

U	Instruction Details	Feature
0	SDOT (4-way, multiple and single vector)	FEAT_SME2
1	UDOT (4-way, multiple and single vector)	FEAT_SME2

SME2 single-multi ternary FP four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		1	1	0	Zn				0	S	off3		

S	Instruction Details	Feature
0	FMLA (multiple and single vector)	FEAT_SME2
1	FMLS (multiple and single vector)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi ternary int four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2–0
1 1 0 0 0 0 0 1 0	sz	1 1	Zm	0	Rv	1 1 0	Zn	1	S	off3

S	Instruction Details	Feature
0	ADD (to array, multiple and single vector)	FEAT_SME2
1	SUB (to array, multiple and single vector)	FEAT_SME2

SME2 single-multi ternary FP16 four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2–0
1 1 0 0 0 0 0 1 0	sz	1 1	Zm	0	Rv	1 1 1	Zn	0	S	off3

sz	S	Instruction Details	Feature
0	0	FMLA (multiple and single vector)	FEAT_SME_F16F16
0	1	FMLS (multiple and single vector)	FEAT_SME_F16F16
1	0	BFMLA (multiple and single vector)	FEAT_SME_B16B16
1	1	BFMLS (multiple and single vector)	FEAT_SME_B16B16

SME2 Multi-vector - Multiple Array Vectors (Two registers)

SME2 Multi-vector - Multiple Array Vectors (Two registers)

The encodings in this section are decoded from SME encodings.

31–23	22	21	20–19	18–17	16–15	14–13	12–10	9–6	5	4–2	1	0
1 1 0 0 0 0 0 1 1	op0	1	op1	op2	0 0		op3		op4	op5	op6	

op0	op1	op2	op3	op4	op5	op6	Instruction details	Feature
0	xx	xx	000	1	000	0	FMLALL (multiple vectors) — Two ZA quad-vectors	FEAT_SME_F8F32

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	op5	op6	Instruction details	Feature
0	xx	xx	000	1	001	0	UNALLOCATED	-
0	xx	xx	000	1	10x	0	UNALLOCATED	-
0	xx	xx	000	1	x0x	1	UNALLOCATED	-
0	xx	xx	010	0	xx0	x	SME2 multiple vectors long FMA two sources	-
0	xx	xx	010	1	000	x	FMLAL (multiple vectors, FP8 to FP16) — Two ZA double-vectors	FEAT_SME_F8F16
0	xx	xx	010	1	100	x	UNALLOCATED	-
0	xx	xx	010	1	x01	x	UNALLOCATED	-
0	xx	xx	0x1	1	xxx	x	UNALLOCATED	-
0	xx	xx	100	x	x0x	x	SME2 multiple vectors FP dot product two registers	-
0	xx	xx	101	0	01x	x	USDOT (4-way, multiple vectors) — Two ZA single-vectors	FEAT_SME2
0	xx	xx	101	0	11x	x	UNALLOCATED	-
0	xx	xx	110	1	x0x	x	UNALLOCATED	-
0	xx	xx	1x1	1	xxx	x	UNALLOCATED	-
0	xx	xx	xx0	1	x1x	x	UNALLOCATED	-
1	xx	xx	010	0	xx0	x	SME2 multiple vectors long MLA two sources	-
1	xx	xx	100	0	x0x	x	UNALLOCATED	-
1	xx	xx	101	0	x1x	x	SME2 multiple vectors two-way dot product two registers	-
1	xx	xx	xxx	1	xxx	x	UNALLOCATED	-
x	00	00	111	0	0xx	x	SME2 multiple vectors binary FP two registers	-
x	00	00	111	0	1xx	x	SME2 multiple vectors binary int two registers	-
x	00	10	111	0	0xx	x	SME2 multiple vectors binary FP16 two registers	-
x	00	10	111	0	1xx	x	UNALLOCATED	-
x	00	x1	111	0	xxx	x	UNALLOCATED	-
x	xx	xx	000	0	xxx	0	SME2 multiple vectors long long MLA two sources	-
x	xx	xx	000	0	xxx	1	UNALLOCATED	-
x	xx	xx	010	0	xx1	x	UNALLOCATED	-
x	xx	xx	0x1	0	xxx	x	UNALLOCATED	-
x	xx	xx	100	0	x1x	x	SME2 multiple vectors ternary FP16 two registers	-
x	xx	xx	101	0	x0x	x	SME2 multiple vectors four-way dot product two registers	-
x	xx	xx	110	0	0xx	x	SME2 multiple vectors ternary FP two registers	-
x	xx	xx	110	0	1xx	x	SME2 multiple vectors ternary int two registers	-
x	!= 00	xx	111	0	xxx	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors long FMA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2	1		0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv			0	1		0	Zn				0	op	S	0	off2		

op	S	Instruction Details	Feature
0	0	FMLAL (multiple vectors, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple vectors)	FEAT_SME2
1	0	BFMLAL (multiple vectors)	FEAT_SME2
1	1	BFMLSL (multiple vectors)	FEAT_SME2

SME2 multiple vectors FP dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5		4	3	2			0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv			1	0		0	Zn				opc			0	off3		

opc	Instruction Details	Feature
00	FDOT (2-way, multiple vectors, FP16 to FP32)	FEAT_SME2
01	BFDOT (multiple vectors)	FEAT_SME2
10	FDOT (2-way, multiple vectors, FP8 to FP16)	FEAT_SME_F8F16
11	FDOT (4-way, multiple vectors)	FEAT_SME_F8F32

SME2 multiple vectors long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2	1		0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv			0	1		0	Zn				0	U	S	0	off2		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	S	Instruction Details	Feature
0	0	SMLAL (multiple vectors)	FEAT_SME2
0	1	SMLSL (multiple vectors)	FEAT_SME2
1	0	UMLAL (multiple vectors)	FEAT_SME2
1	1	UMLSL (multiple vectors)	FEAT_SME2

SME2 multiple vectors two-way dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv			1	0		1	Zn				0	U	1	off3			

U	Instruction Details	Feature
0	SDOT (2-way, multiple vectors)	FEAT_SME2
1	UDOT (2-way, multiple vectors)	FEAT_SME2

SME2 multiple vectors binary FP two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20		19	18		17	16		15	14		13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1	0		0	0		0	0		0	Rv			1	1		1	Zm				0	0	S	off3			

S	Instruction Details	Feature
0	FADD	FEAT_SME2
1	FSUB	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors binary int two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv		1	1	1	Zm			0	1	S	off3		

S	Instruction Details	Feature
0	ADD (to array, array and multiple vectors)	FEAT_SME2
1	SUB (to array, array and multiple vectors)	FEAT_SME2

SME2 multiple vectors binary FP16 two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	1	0	0	0	Rv		1	1	1	Zm			0	0	S	off3		

sz	S	Instruction Details	Feature
0	0	FADD	FEAT_SME_F16F16 || FEAT_SME_F8F16
0	1	FSUB	FEAT_SME_F16F16 || FEAT_SME_F8F16
1	0	BFADD	FEAT_SME_B16B16
1	1	BFSUB	FEAT_SME_B16B16

SME2 multiple vectors long long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14	13	12		10	9		6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv		0	0	0	Zn			0	U	S	op	0	o1

sz	U	S	op	Instruction Details	Feature
x	0	0	0	SMLALL (multiple vectors)	FEAT_SME2
x	0	1	0	SMLSLL (multiple vectors)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	U	S	op	Instruction Details	Feature
x	1	0	0	UMLALL (multiple vectors)	FEAT_SME2
x	1	1	0	UMLSLL (multiple vectors)	FEAT_SME2
0	0	0	1	USMLALL (multiple vectors)	FEAT_SME2
0	0	1	1	UNALLOCATED	-
0	1	x	1	UNALLOCATED	-
1	x	x	1	UNALLOCATED	-

SME2 multiple vectors ternary FP16 two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv			1	0		0	Zn				0	S	1	off3			

sz	S	Instruction Details	Feature
0	0	FMLA (multiple vectors)	FEAT_SME_F16F16
0	1	FMLS (multiple vectors)	FEAT_SME_F16F16
1	0	BFMLA (multiple vectors)	FEAT_SME_B16B16
1	1	BFMLS (multiple vectors)	FEAT_SME_B16B16

SME2 multiple vectors four-way dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv			1	0		1	Zn				0	U	0	off3			

U	Instruction Details	Feature
0	SDOT (4-way, multiple vectors)	FEAT_SME2
1	UDOT (4-way, multiple vectors)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors ternary FP two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14	13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	1		1	0			Zn		0	0	S		off3		

S	Instruction Details	Feature
0	FMLA (multiple vectors)	FEAT_SME2
1	FMLS (multiple vectors)	FEAT_SME2

SME2 multiple vectors ternary int two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14	13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	1		1	0			Zn		0	1	S		off3		

S	Instruction Details	Feature
0	ADD (to array, multiple vectors)	FEAT_SME2
1	SUB (to array, multiple vectors)	FEAT_SME2

SME2 Multi-vector - Multiple Array Vectors (Four registers)

SME2 Multi-vector - Multiple Array Vectors (Four registers)

The encodings in this section are decoded from SME encodings.

31								23	22	21	20	19	18	17	16	15	14	13	12			10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	1	op0	1	op1		op2		1	0			op3								op4		op5				op6	

op0	op1	op2	op3	op4	op5	op6	Instruction details	Feature
0	xx	x0	000	01	000	0	FMLALL (multiple vectors) — Four ZA quad-vectors	FEAT_SME_F8F32
0	xx	x0	000	01	001	0	UNALLOCATED	-
0	xx	x0	000	01	10x	0	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	op5	op6	Instruction details	Feature
0	xx	x0	000	01	x0x	1	UNALLOCATED	-
0	xx	x0	010	00	xx0	x	SME2 multiple vectors long FMA four sources	-
0	xx	x0	010	01	000	x	FMLAL (multiple vectors, FP8 to FP16) — Four ZA double-vectors	FEAT_SME_F8F16
0	xx	x0	010	01	100	x	UNALLOCATED	-
0	xx	x0	010	01	x01	x	UNALLOCATED	-
0	xx	x0	0x1	01	xxx	x	UNALLOCATED	-
0	xx	x0	100	0x	x0x	x	SME2 multiple vectors FP dot product four registers	-
0	xx	x0	101	00	01x	x	USDOT (4-way, multiple vectors) — Four ZA single-vectors	FEAT_SME2
0	xx	x0	101	00	11x	x	UNALLOCATED	-
0	xx	x0	110	01	x0x	x	UNALLOCATED	-
0	xx	x0	1x1	01	xxx	x	UNALLOCATED	-
0	xx	x0	xx0	01	x1x	x	UNALLOCATED	-
1	xx	x0	010	00	xx0	x	SME2 multiple vectors long MLA four sources	-
1	xx	x0	100	00	x0x	x	UNALLOCATED	-
1	xx	x0	101	00	x1x	x	SME2 multiple vectors two-way dot product four registers	-
1	xx	x0	xxx	01	xxx	x	UNALLOCATED	-
x	00	00	111	00	0xx	x	SME2 multiple vectors binary FP four registers	-
x	00	00	111	00	1xx	x	SME2 multiple vectors binary int four registers	-
x	00	10	111	00	0xx	x	SME2 multiple vectors binary FP16 four registers	-
x	00	10	111	00	1xx	x	UNALLOCATED	-
x	xx	x0	000	00	xxx	0	SME2 multiple vectors long long MLA four sources	-
x	xx	x0	000	00	xxx	1	UNALLOCATED	-
x	xx	x0	010	00	xx1	x	UNALLOCATED	-
x	xx	x0	0x1	00	xxx	x	UNALLOCATED	-
x	xx	x0	100	00	x1x	x	SME2 multiple vectors ternary FP16 four registers	-
x	xx	x0	101	00	x0x	x	SME2 multiple vectors four-way dot product four registers	-
x	xx	x0	110	00	0xx	x	SME2 multiple vectors ternary FP four registers	-
x	xx	x0	110	00	1xx	x	SME2 multiple vectors ternary int four registers	-
x	xx	x0	xxx	1x	xxx	x	UNALLOCATED	-
x	xx	x1	xxx	xx	xxx	x	UNALLOCATED	-
x	!= 00	x0	111	00	xxx	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors long FMA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	op	S	0	off2	

op	S	Instruction Details	Feature
0	0	FMLAL (multiple vectors, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple vectors)	FEAT_SME2
1	0	BFMLAL (multiple vectors)	FEAT_SME2
1	1	BFMLSL (multiple vectors)	FEAT_SME2

SME2 multiple vectors FP dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5		4	3	2		0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		1	0	0	Zn			0	opc			0	off3		

opc	Instruction Details	Feature
00	FDOT (2-way, multiple vectors, FP16 to FP32)	FEAT_SME2
01	BFDOT (multiple vectors)	FEAT_SME2
10	FDOT (2-way, multiple vectors, FP8 to FP16)	FEAT_SME_F8F16
11	FDOT (4-way, multiple vectors)	FEAT_SME_F8F32

SME2 multiple vectors long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	U	S	0	off2	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	S	Instruction Details	Feature
0	0	SMLAL (multiple vectors)	FEAT_SME2
0	1	SMLSL (multiple vectors)	FEAT_SME2
1	0	UMLAL (multiple vectors)	FEAT_SME2
1	1	UMLSL (multiple vectors)	FEAT_SME2

SME2 multiple vectors two-way dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	1	0	Rv		1	0	1	Zn			0	0	U	1	off3		

U	Instruction Details	Feature
0	SDOT (2-way, multiple vectors)	FEAT_SME2
1	UDOT (2-way, multiple vectors)	FEAT_SME2

SME2 multiple vectors binary FP four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		19	18		17	16		15	14	13	12			10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0		0	0		0	1		0	Rv		1		1	1	Zm			0	0	0	S	off3		

S	Instruction Details	Feature
0	FADD	FEAT_SME2
1	FSUB	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors binary int four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20	19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv		1	1	1	Zm			0	0	1	S	off3		

S	Instruction Details	Feature
0	ADD (to array, array and multiple vectors)	FEAT_SME2
1	SUB (to array, array and multiple vectors)	FEAT_SME2

SME2 multiple vectors binary FP16 four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20	19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	1	0	1	0	Rv		1	1	1	Zm			0	0	0	S	off3		

sz	S	Instruction Details	Feature
0	0	FADD	FEAT_SME_F16F16 || FEAT_SME_F8F16
0	1	FSUB	FEAT_SME_F16F16 || FEAT_SME_F8F16
1	0	BFADD	FEAT_SME_B16B16
1	1	BFSUB	FEAT_SME_B16B16

SME2 multiple vectors long long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm			0	1	0	Rv		0	0	0	Zn			0	0	U	S	op	0	o1

sz	U	S	op	Instruction Details	Feature
x	0	0	0	SMLALL (multiple vectors)	FEAT_SME2
x	0	1	0	SMLSLL (multiple vectors)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	U	S	op	Instruction Details	Feature
x	1	0	0	UMLALL (multiple vectors)	FEAT_SME2
x	1	1	0	UMLSLL (multiple vectors)	FEAT_SME2
0	0	0	1	USMLALL (multiple vectors)	FEAT_SME2
0	0	1	1	UNALLOCATED	-
0	1	x	1	UNALLOCATED	-
1	x	x	1	UNALLOCATED	-

SME2 multiple vectors ternary FP16 four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	Zm			0	1	0	Rv		1	0	0	Zn			0	0	S	1	off3		

sz	S	Instruction Details	Feature
0	0	FMLA (multiple vectors)	FEAT_SME_F16F16
0	1	FMLS (multiple vectors)	FEAT_SME_F16F16
1	0	BFMLA (multiple vectors)	FEAT_SME_B16B16
1	1	BFMLS (multiple vectors)	FEAT_SME_B16B16

SME2 multiple vectors four-way dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	Zm			0	1	0	Rv		1	0	1	Zn			0	0	U	0	off3		

U	Instruction Details	Feature
0	SDOT (4-way, multiple vectors)	FEAT_SME2
1	UDOT (4-way, multiple vectors)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors ternary FP four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31	23	22	21	20	18	17	15	14	13	12	10	9	7	6	5	4	3	2	0
1 1 0 0 0 0 0 1 1		sz	1	Zm		0 1 0		Rv		1 1 0		Zn		0 0		0	S	off3	

S	Instruction Details	Feature
0	FMLA (multiple vectors)	FEAT_SME2
1	FMLS (multiple vectors)	FEAT_SME2

SME2 multiple vectors ternary int four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31	23	22	21	20	18	17	15	14	13	12	10	9	7	6	5	4	3	2	0
1 1 0 0 0 0 0 1 1		sz	1	Zm		0 1 0		Rv		1 1 0		Zn		0 0		1	S	off3	

S	Instruction Details	Feature
0	ADD (to array, multiple vectors)	FEAT_SME2
1	SUB (to array, multiple vectors)	FEAT_SME2

SME2 Multi-vector - Indexed (One register)

SME2 Multi-vector - Indexed (One register)

The encodings in this section are decoded from SME encodings.

31	24	23	22	21	20	19	13	12	11	5	4	2	1	0
1 1 0 0 0 0 0	1	op0		0	0			op1			op2			

op0	op1	op2	Instruction details	Feature
00	x	xxx	SME2 multi-vec indexed long long MLA one source 32-bit	-
01	x	000	FMLALL (multiple and indexed vector) — One ZA quad-vector	FEAT_SME_F8F32
01	x	!=000	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details	Feature
10	0	xx0	SME2 multi-vec indexed long long MLA one source 64-bit	-
10	0	xx1	UNALLOCATED	-
10	1	xxx	SME2 multi-vec indexed long FMA one source	-
11	0	0xx	FMLAL (multiple and indexed vector, FP8 to FP16) — One ZA double-vector	FEAT_SME_F8F16
11	0	1xx	UNALLOCATED	-
11	1	xxx	SME2 multi-vec indexed long MLA one source	-

SME2 multi-vec indexed long long MLA one source 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (One register).

31							24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm		i4h	Rv		i4l		Zn		U	S	op	off2	

U	S	op	Instruction Details	Feature
x	1	1	UNALLOCATED	-
0	0	0	SMLALL (multiple and indexed vector)	FEAT_SME2
0	0	1	USMLALL (multiple and indexed vector)	FEAT_SME2
0	1	0	SMLSLL (multiple and indexed vector)	FEAT_SME2
1	0	0	UMLALL (multiple and indexed vector)	FEAT_SME2
1	0	1	SUMLALL (multiple and indexed vector)	FEAT_SME2
1	1	0	UMLSLL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed long long MLA one source 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (One register).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm		i3h	Rv		0	i3l		Zn		U	S	0	off2	

U	S	Instruction Details	Feature
0	0	SMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	S	Instruction Details	Feature
0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	0	UMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed long FMA one source

The encodings in this section are decoded from SME2 Multi-vector - Indexed (One register).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm		i3h	Rv		1	i3l		Zn		op	S	off3	

op	S	Instruction Details	Feature
0	0	FMLAL (multiple and indexed vector, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple and indexed vector)	FEAT_SME2
1	0	BFMLAL (multiple and indexed vector)	FEAT_SME2
1	1	BFMLSL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed long MLA one source

The encodings in this section are decoded from SME2 Multi-vector - Indexed (One register).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	1	1	0	0	Zm		i3h	Rv		1	i3l		Zn		U	S	off3	

U	S	Instruction Details	Feature
0	0	SMLAL (multiple and indexed vector)	FEAT_SME2
0	1	SMLSL (multiple and indexed vector)	FEAT_SME2
1	0	UMLAL (multiple and indexed vector)	FEAT_SME2
1	1	UMLSL (multiple and indexed vector)	FEAT_SME2

SME2 Multi-vector - Indexed (Two registers)

SME2 Multi-vector - Indexed (Two registers)

The encodings in this section are decoded from SME encodings.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20	19			16	15	14	13	12	11	10			6	5	4	3	2		0
1	1	0	0	0	0	0	1	op0		0	1					0			op1						op2	op3				

op0	op1	op2	op3	Instruction details	Feature
00	0x	x	xx	SME2 multi-vec indexed long long MLA two sources 32-bit	-
00	1x	x	xx	SME2 multi-vec ternary indexed two registers 16-bit	-
01	xx	x	xx	SME2 multi-vec ternary indexed two registers 32-bit	-
10	00	0	xx	SME2 multi-vec indexed long long MLA two sources 64-bit	-
10	01	0	xx	UNALLOCATED	-
10	0x	1	00	FMLALL (multiple and indexed vector) — Two ZA quad-vectors	FEAT_SME_F8F32
10	0x	1	!=00	UNALLOCATED	-
10	1x	0	xx	SME2 multi-vec indexed long FMA two sources	-
10	1x	1	0x	UNALLOCATED	-
10	1x	1	1x	FMLAL (multiple and indexed vector, FP8 to FP16) — Two ZA double-vectors	FEAT_SME_F8F16
11	00	0	xx	SME2 multi-vec ternary indexed two registers 64-bit	-
11	01	0	xx	SME2 multi-vec indexed FP8 two-way vertical dot product to single-precision two registers	-
11	1x	0	xx	SME2 multi-vec indexed long MLA two sources	-
11	xx	1	0x	SME2 multi-vec indexed FP8 two-way dot product to FP16 two registers	-
11	xx	1	1x	UNALLOCATED	-

SME2 multi-vec indexed long long MLA two sources 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1			Zm		0		Rv	0		i4h			Zn		op	U	S		i4l	o1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	S	Instruction Details	Feature
0	0	0	SMLALL (multiple and indexed vector)	FEAT_SME2
0	0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2
0	1	0	UMLALL (multiple and indexed vector)	FEAT_SME2
0	1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2
1	x	1	UNALLOCATED	-
1	0	0	USMLALL (multiple and indexed vector)	FEAT_SME2
1	1	0	SUMLALL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec ternary indexed two registers 16-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm		0	Rv		1	i3h		Zn		op	S	i3l	off3	

op	S	Instruction Details	Feature
0	0	FMLA (multiple and indexed vector)	FEAT_SME_F16F16
0	1	FMLS (multiple and indexed vector)	FEAT_SME_F16F16
1	0	BFMLA (multiple and indexed vector)	FEAT_SME_B16B16
1	1	BFMLS (multiple and indexed vector)	FEAT_SME_B16B16

SME2 multi-vec ternary indexed two registers 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	3	2	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm		0	Rv		op	i2		Zn		opc2		off3	

op	opc2	Instruction Details	Feature
0	000	FMLA (multiple and indexed vector)	FEAT_SME2
0	001	FVDOT (FP16 to FP32)	FEAT_SME2
0	010	FMLS (multiple and indexed vector)	FEAT_SME2
0	011	BFVDOT	FEAT_SME2
0	100	SVDOT (2-way)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	opc2	Instruction Details	Feature
0	101	UNALLOCATED	-
0	110	UVDOT (2-way)	FEAT_SME2
0	111	FDOT (4-way, multiple and indexed vector)	FEAT_SME_F8F32
1	000	SDOT (2-way, multiple and indexed vector)	FEAT_SME2
1	001	FDOT (2-way, multiple and indexed vector, FP16 to FP32)	FEAT_SME2
1	010	UDOT (2-way, multiple and indexed vector)	FEAT_SME2
1	011	BFDOT (multiple and indexed vector)	FEAT_SME2
1	100	SDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	101	USDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	110	UDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	111	SUDOT (4-way, multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed long long MLA two sources 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm		0	Rv		0	0	i3h	Zn		0	U	S	i3l		o1

U	S	Instruction Details	Feature
0	0	SMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	0	UMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed long FMA two sources

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm		0	Rv		1	i3h		Zn		0	op	S	i3l		off2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	Instruction Details	Feature
0	0	FMLAL (multiple and indexed vector, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple and indexed vector)	FEAT_SME2
1	0	BFMLAL (multiple and indexed vector)	FEAT_SME2
1	1	BFMLSL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec ternary indexed two registers 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19			16	15	14		13	12		11	10	9			6	5	4		3	2			0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv			0		0	i1	Zn				0	opc			off3			

opc	Instruction Details	Feature
00	FMLA (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_F64F64
01	SDOT (4-way, multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
10	FMLS (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_F64F64
11	UDOT (4-way, multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed FP8 two-way vertical dot product to single-precision two registers

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19			16	15	14		13	12		11	10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv			0		1	i2h	Zn				0	T	i2l	off3			

T	Instruction Details	Feature
0	FVDOTB	FEAT_SME_F8F32
1	FVDOTT	FEAT_SME_F8F32

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec indexed long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm		0	Rv		1	i3h		Zn		0	U	S	i3l	off2	

U	S	Instruction Details	Feature
0	0	SMLAL (multiple and indexed vector)	FEAT_SME2
0	1	SMLSL (multiple and indexed vector)	FEAT_SME2
1	0	UMLAL (multiple and indexed vector)	FEAT_SME2
1	1	UMLSL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed FP8 two-way dot product to FP16 two registers

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm		0	Rv		op	i3h		Zn		1	0	i3l	off3	

op	Instruction Details	Feature
0	FDOT (2-way, multiple and indexed vector, FP8 to FP16)	FEAT_SME_F8F16
1	FVDOT (FP8 to FP16)	FEAT_SME_F8F16

SME2 Multi-vector - Indexed (Four registers)

SME2 Multi-vector - Indexed (Four registers)

The encodings in this section are decoded from SME encodings.

31							24	23	22	21	20	19	16	15	14	13	12	11	10	7	6	4	3	2	0
1	1	0	0	0	0	0	1	op0		0	1			1			op1				op2		op3		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
00	0x	0xx	x	SME2 multi-vec indexed long long MLA four sources 32-bit	-
00	0x	100	0	FMLALL (multiple and indexed vector) — Four ZA quad-vectors	FEAT_SME_F8F32
00	0x	100	1	UNALLOCATED	-
00	1x	0xx	x	SME2 multi-vec ternary indexed four registers 16-bit	-
00	1x	100	x	FDOT (2-way, multiple and indexed vector, FP8 to FP16) — Four ZA single-vectors	FEAT_SME_F8F16
00	xx	101	x	UNALLOCATED	-
00	xx	11x	x	UNALLOCATED	-
01	xx	0xx	x	SME2 multi-vec ternary indexed four registers 32-bit	-
10	00	00x	x	SME2 multi-vec indexed long long MLA four sources 64-bit	-
10	01	00x	x	UNALLOCATED	-
10	0x	01x	x	UNALLOCATED	-
10	1x	00x	x	SME2 multi-vec indexed long FMA four sources	-
10	1x	010	x	FMLAL (multiple and indexed vector, FP8 to FP16) — Four ZA double-vectors	FEAT_SME_F8F16
10	1x	011	x	UNALLOCATED	-
11	0x	00x	x	SME2 multi-vec ternary indexed four registers 64-bit	-
11	1x	00x	x	SME2 multi-vec indexed long MLA four sources	-
11	xx	01x	x	UNALLOCATED	-
!=00	xx	1xx	x	UNALLOCATED	-

SME2 multi-vec indexed long long MLA four sources 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		0	i4h		Zn				0	op	U	S	i4l		o1

op	U	S	Instruction Details	Feature
0	0	0	SMLALL (multiple and indexed vector)	FEAT_SME2
0	0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2
0	1	0	UMLALL (multiple and indexed vector)	FEAT_SME2
0	1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	S	Instruction Details	Feature
1	x	1	UNALLOCATED	-
1	0	0	USMLALL (multiple and indexed vector)	FEAT_SME2
1	1	0	SUMLALL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec ternary indexed four registers 16-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19			16	15	14		13	12	11		10	9			7	6	5		4	3			2			0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv			1	i3h			Zn				0	op		S	i3l			off3		

op	S	Instruction Details	Feature
0	0	FMLA (multiple and indexed vector)	FEAT_SME_F16F16
0	1	FMLS (multiple and indexed vector)	FEAT_SME_F16F16
1	0	BFMLA (multiple and indexed vector)	FEAT_SME_B16B16
1	1	BFMLS (multiple and indexed vector)	FEAT_SME_B16B16

SME2 multi-vec ternary indexed four registers 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19			16	15	14		13	12	11		10	9			7	6	5			3	2			0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv			op	i2			Zn				0	opc2				off3			

op	opc2	Instruction Details	Feature
0	000	FMLA (multiple and indexed vector)	FEAT_SME2
0	001	FDOT (4-way, multiple and indexed vector)	FEAT_SME_F8F32
0	010	FMLS (multiple and indexed vector)	FEAT_SME2
0	011	UNALLOCATED	-
0	100	SVDOT (4-way)	FEAT_SME2
0	101	USVDOT	FEAT_SME2
0	110	UVDOT (4-way)	FEAT_SME2
0	111	SUVDOT	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	opc2	Instruction Details	Feature
1	000	SDOT (2-way, multiple and indexed vector)	FEAT_SME2
1	001	FDOT (2-way, multiple and indexed vector, FP16 to FP32)	FEAT_SME2
1	010	UDOT (2-way, multiple and indexed vector)	FEAT_SME2
1	011	BFDOT (multiple and indexed vector)	FEAT_SME2
1	100	SDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	101	USDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	110	UDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	111	SUDOT (4-way, multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed long long MLA four sources 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19			16	15	14		13	12		11	10	9			7	6		5	4	3	2		1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				1	Rv			0		0	i3h	Zn				0		0	U	S	i3l			o1

U	S	Instruction Details	Feature
0	0	SMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	0	UMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed long FMA four sources

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19			16	15	14		13	12	11		10	9			7	6		5	4	3	2	1		0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				1	Rv			1	i3h			Zn				0		0	op	S	i3l			off2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	Instruction Details	Feature
0	0	FMLAL (multiple and indexed vector, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple and indexed vector)	FEAT_SME2
1	0	BFMLAL (multiple and indexed vector)	FEAT_SME2
1	1	BFMLSL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec ternary indexed four registers 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19			16	15	14		13	12	11	10	9			7	6	5	4		3	2			0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv			0	op	i1	Zn				0	0	opc2			off3			

op	opc2	Instruction Details	Feature
0	00	FMLA (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_F64F64
0	01	SDOT (4-way, multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
0	10	FMLS (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_F64F64
0	11	UDOT (4-way, multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	x0	UNALLOCATED	-
1	01	SVDOT (4-way)	FEAT_SME2 && FEAT_SME_I16I64
1	11	UVDOT (4-way)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19			16	15	14		13	12	11		10	9			7	6	5	4	3	2	1		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv			1	i3h			Zn				0	0	U	S	i3l	off2		

U	S	Instruction Details	Feature
0	0	SMLAL (multiple and indexed vector)	FEAT_SME2
0	1	SMLSL (multiple and indexed vector)	FEAT_SME2
1	0	UMLAL (multiple and indexed vector)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	S	Instruction Details	Feature
1	1	UMLSL (multiple and indexed vector)	FEAT_SME2

SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers)

SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers)

The encodings in this section are decoded from SME encodings.

31	24	23	22	21	20	19	16	15	11	10	9	8	7	5	4	0
1 1 0 0 0 0 0 1				1	0			1 0 1 0 0		op0	op1		op2			

op0	op1	op2	Instruction details
0	00	00x	SME2 single-multi int min/max two registers
0	00	101	UNALLOCATED
0	01	00x	SME2 single-multi FP min/max two registers
0	01	100	SME2 multiple and single vector FSCALE two registers
0	01	101	UNALLOCATED
0	0x	x1x	UNALLOCATED
0	10	xxx	SME2 single-multi shift two registers
0	11	000	SME2 single-multi add two registers
0	11	!= 000	UNALLOCATED
1	00	000	SME2 single-multi signed saturating doubling multiply high two registers
1	00	x10	UNALLOCATED
1	00	xx1	UNALLOCATED
1	!= 00	xxx	UNALLOCATED
x	00	100	UNALLOCATED

SME2 single-multi int min/max two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31	24	23	22	21	20	19	16	15	11	10	9	8	7	6	5	4	1	0
1 1 0 0 0 0 0 1		size		1	0	Zm		1 0 1 0 0		0	0	0	0	0	op	Zdn		U

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	Instruction Details	Feature
0	0	SMAX (multiple and single vector)	FEAT_SME2
0	1	UMAX (multiple and single vector)	FEAT_SME2
1	0	SMIN (multiple and single vector)	FEAT_SME2
1	1	UMIN (multiple and single vector)	FEAT_SME2

SME2 single-multi FP min/max two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31							24	23 22	21	20	19 16	15				11	10	9	8	7	6	5	4 1	0
1	1	0	0	0	0	0	1	size	1	0	Zm	1	0	1	0	0	0	0	1	0	0	op	Zdn	o2

size	op	o2	Instruction Details	Feature
00	0	0	BFMAX (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	0	1	BFMIN (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	1	0	BFMAXNM (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	1	1	BFMINNM (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	0	0	FMAX (multiple and single vector)	FEAT_SME2
!= 00	0	1	FMIN (multiple and single vector)	FEAT_SME2
!= 00	1	0	FMAXNM (multiple and single vector)	FEAT_SME2
!= 00	1	1	FMINNM (multiple and single vector)	FEAT_SME2

SME2 multiple and single vector FSCALE two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31							24	23 22	21	20	19 16	15				11	10	9	8	7		5	4 1	0
1	1	0	0	0	0	0	1	size	1	0	Zm	1	0	1	0	0	0	0	1	1	0	0	Zdn	op

size	op	Instruction Details	Feature
xx	1	UNALLOCATED	-
00	0	BFSCALE (multiple and single vector)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	0	FSCALE (multiple and single vector)	FEAT_SME2 && FEAT_FP8

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi shift two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31							24	23	22	21	20	19			16	15					11	10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0		0	1	0	opc				Zdn				U

opc	U	Instruction Details	Feature
001	0	SRSHL (multiple and single vector)	FEAT_SME2
001	1	URSHL (multiple and single vector)	FEAT_SME2
!=001	x	UNALLOCATED	-

SME2 single-multi add two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31							24	23	22	21	20	19			16	15					11	10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0		0	1	1	0	0		0	Zdn				op

op	Instruction Details	Feature
0	ADD (to vector, multiple vectors)	FEAT_SME2
1	UNALLOCATED	-

SME2 single-multi signed saturating doubling multiply high two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31							24	23	22	21	20	19			16	15					11	10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0		1	0	0	0	0		0	Zdn				op

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Feature
0	SQDMULH (multiple and single vector)	FEAT_SME2
1	UNALLOCATED	-

SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers)

SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers)

The encodings in this section are decoded from SME encodings.

31							24	23	22	21	20	19				16	15				11	10	9	8	7			5	4			2	1	0
1	1	0	0	0	0	0	1			1	0						1	0	1	0	1	op0	op1		op2								0	

op0	op1	op2	Instruction details
0	00	00x	SME2 single-multi int min/max four registers
0	00	101	UNALLOCATED
0	01	00x	SME2 single-multi FP min/max four registers
0	01	100	SME2 multiple and single vector FSCALE four registers
0	01	101	UNALLOCATED
0	0x	x1x	UNALLOCATED
0	10	xxx	SME2 single-multi shift four registers
0	11	000	SME2 single-multi add four registers
0	11	!= 000	UNALLOCATED
1	00	000	SME2 single-multi signed saturating doubling multiply high four registers
1	00	x10	UNALLOCATED
1	00	xx1	UNALLOCATED
1	!= 00	xxx	UNALLOCATED
x	00	100	UNALLOCATED

SME2 single-multi int min/max four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23	22	21	20	19				16	15				11	10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm					1	0	1	0	1	0	0	0	0	0	op	Zdn				0	U

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	Instruction Details	Feature
0	0	SMAX (multiple and single vector)	FEAT_SME2
0	1	UMAX (multiple and single vector)	FEAT_SME2
1	0	SMIN (multiple and single vector)	FEAT_SME2
1	1	UMIN (multiple and single vector)	FEAT_SME2

SME2 single-multi FP min/max four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23 22	21	20	19 16	15				11	10	9	8	7	6	5	4 2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm	1	0	1	0	1	0	0	1	0	0	op	Zdn	0	o2

size	op	o2	Instruction Details	Feature
00	0	0	BFMAX (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	0	1	BFMIN (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	1	0	BFMAXNM (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	1	1	BFMINNM (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	0	0	FMAX (multiple and single vector)	FEAT_SME2
!= 00	0	1	FMIN (multiple and single vector)	FEAT_SME2
!= 00	1	0	FMAXNM (multiple and single vector)	FEAT_SME2
!= 00	1	1	FMINNM (multiple and single vector)	FEAT_SME2

SME2 multiple and single vector FSCALE four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23 22	21	20	19 16	15				11	10	9	8	7		5	4 2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm	1	0	1	0	1	0	0	1	1	0	0	Zdn	0	op

size	op	Instruction Details	Feature
xx	1	UNALLOCATED	-
00	0	BFSCALE (multiple and single vector)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	0	FSCALE (multiple and single vector)	FEAT_SME2 && FEAT_FP8

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi shift four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23	22	21	20	19			16	15				11	10	9	8	7			5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	1	0	opc				Zdn				0	U

opc	U	Instruction Details	Feature
001	0	SRSHL (multiple and single vector)	FEAT_SME2
001	1	URSHL (multiple and single vector)	FEAT_SME2
!=001	x	UNALLOCATED	-

SME2 single-multi add four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23	22	21	20	19			16	15				11	10	9	8	7			5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	1	1	0	0	0		Zdn				0	op

op	Instruction Details	Feature
0	ADD (to vector, multiple vectors)	FEAT_SME2
1	UNALLOCATED	-

SME2 single-multi signed saturating doubling multiply high four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23	22	21	20	19			16	15				11	10	9	8	7			5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	1	0	0	0	0	0		Zdn				0	op

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Feature
0	SQDMULH (multiple and single vector)	FEAT_SME2
1	UNALLOCATED	-

SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers)

SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers)

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20		18	17							10	9		7	6		2	1	0
1	1	0	0	0	0	0	1			1				0	0	1	0	1	1	1	0	op0						0	

op0	Instruction details
000	SME2 multiple vectors int min/max four registers
001	UNALLOCATED
010	SME2 multiple vectors FP min/max four registers
011	SME2 multiple vectors FSCALE four registers
10x	SME2 multiple vectors shift four registers
11x	UNALLOCATED

SME2 multiple vectors int min/max four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers).

31							24	23	22	21	20		18	17							10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm			0	0	1	0	1	1	1	0	0	0	0	opc		Zdn			0	U

opc	U	Instruction Details	Feature
00	0	SMAX (multiple vectors)	FEAT_SME2
00	1	UMAX (multiple vectors)	FEAT_SME2
01	0	SMIN (multiple vectors)	FEAT_SME2
01	1	UMIN (multiple vectors)	FEAT_SME2
1x	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors FP min/max four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers).

31							24	23	22	21	20			18	17							10	9			7	6		5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	0	1	0	1	1	1	0	0	1	0		opc			Zdn				0	o2

size	opc	o2	Instruction Details	Feature
xx	10	0	FAMAX	FEAT_SME2 && FEAT_FAMINMAX
xx	10	1	FAMIN	FEAT_SME2 && FEAT_FAMINMAX
xx	11	x	UNALLOCATED	-
00	00	0	BFMAX (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	00	1	BFMIN (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	01	0	BFMAXNM (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	01	1	BFMINNM (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	00	0	FMAX (multiple vectors)	FEAT_SME2
!= 00	00	1	FMIN (multiple vectors)	FEAT_SME2
!= 00	01	0	FMAXNM (multiple vectors)	FEAT_SME2
!= 00	01	1	FMINNM (multiple vectors)	FEAT_SME2

SME2 multiple vectors FSCALE four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers).

31							24	23	22	21	20			18	17							10	9			7	6		5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	0	1	0	1	1	1	0	0	1	1		opc			Zdn				0	o2

size	opc	o2	Instruction Details	Feature
xx	00	1	UNALLOCATED	-
xx	!= 00	x	UNALLOCATED	-
00	00	0	BFSCALE (multiple vectors)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	00	0	FSCALE (multiple vectors)	FEAT_SME2 && FEAT_FP8

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors shift four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers).

31							24	23	22	21	20			18	17									8	7			5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	0	1	0	1	1	1	0	1	0	opc				Zdn				0	U

opc	U	Instruction Details	Feature
001	0	SRSHL (multiple vectors)	FEAT_SME2
001	1	URSHL (multiple vectors)	FEAT_SME2
!=001	x	UNALLOCATED	-

SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers)

SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers)

The encodings in this section are decoded from SME encodings.

31							25	24	23	22	21	20			18	17								10	9				5	4			2	1	0
1	1	0	0	0	0	0		1			1					0	0	1	0	1	1	1	1		op0									0	

op0	Instruction details
00000	SME2 multi-vector signed saturating doubling multiply high four registers
!=00000	UNALLOCATED

SME2 multi-vector signed saturating doubling multiply high four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers).

31							24	23	22	21	20			18	17									10	9				5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	0	1	0	1	1	1	1	0	0	0	0	0	0	Zdn				0	op

op	Instruction Details	Feature
0	SQDMULH (multiple vectors)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Feature
1	UNALLOCATED	-

SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers)

SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers)

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20		17	16						10	9		7	6		0
1	1	0	0	0	0	0	1			1				0	1	0	1	1	0	0	op0					

op0	Instruction details
000	SME2 multiple vectors int min/max two registers
001	UNALLOCATED
010	SME2 multiple vectors FP min/max two registers
011	SME2 multiple vectors FSCALE two registers
10x	SME2 multiple vectors shift two registers
11x	UNALLOCATED

SME2 multiple vectors int min/max two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers).

31							24	23	22	21	20		17	16						10	9		7	6	5	4		1	0
1	1	0	0	0	0	0	1	size		1	Zm			0	1	0	1	1	0	0	0	0	0	opc		Zdn			U

opc	U	Instruction Details	Feature
00	0	SMAX (multiple vectors)	FEAT_SME2
00	1	UMAX (multiple vectors)	FEAT_SME2
01	0	SMIN (multiple vectors)	FEAT_SME2
01	1	UMIN (multiple vectors)	FEAT_SME2
1x	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors FP min/max two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers).

31							24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	0	1	0		opc		Zdn				o2

size	opc	o2	Instruction Details	Feature
xx	10	0	FAMAX	FEAT_SME2 && FEAT_FAMINMAX
xx	10	1	FAMIN	FEAT_SME2 && FEAT_FAMINMAX
xx	11	x	UNALLOCATED	-
00	00	0	BFMAX (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	00	1	BFMIN (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	01	0	BFMAXNM (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	01	1	BFMINNM (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	00	0	FMAX (multiple vectors)	FEAT_SME2
!= 00	00	1	FMIN (multiple vectors)	FEAT_SME2
!= 00	01	0	FMAXNM (multiple vectors)	FEAT_SME2
!= 00	01	1	FMINNM (multiple vectors)	FEAT_SME2

SME2 multiple vectors FSCALE two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers).

31							24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	0	1	1		opc		Zdn				o2

size	opc	o2	Instruction Details	Feature
xx	00	1	UNALLOCATED	-
xx	!= 00	x	UNALLOCATED	-
00	00	0	BFSCALE (multiple vectors)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	00	0	FSCALE (multiple vectors)	FEAT_SME2 && FEAT_FP8

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors shift two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers).

31							24	23	22	21	20				17	16							8	7			5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm					0	1	0	1	1	0	0	1	0	opc			Zdn				U

opc	U	Instruction Details	Feature
001	0	SRSHL (multiple vectors)	FEAT_SME2
001	1	URSHL (multiple vectors)	FEAT_SME2
!=001	x	UNALLOCATED	-

SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers)

SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers)

The encodings in this section are decoded from SME encodings.

31							25	24	23	22	21	20				17	16						10	9				5	4				0
1	1	0	0	0	0	0		1			1						0	1	0	1	1	0	1	op0									

op0	Instruction details
00000	SME2 multi-vector signed saturating doubling multiply high two registers
!=00000	UNALLOCATED

SME2 multi-vector signed saturating doubling multiply high two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers).

31							24	23	22	21	20				17	16							10	9				5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm					0	1	0	1	1	0	1		0	0	0	0	0	Zdn				op

op	Instruction Details	Feature
0	SQDMULH (multiple vectors)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Feature
1	UNALLOCATED	-

SME2 Multi-vector - SVE Select

SME2 Multi-vector - SVE Select

The encodings in this section are decoded from SME encodings.

31	24	23	22	21	20	18	17	16	15	13	12	7	6	5	4	2	1	0
1 1 0 0 0 0 0 1				1			op0		1 0 0				op1				op2	

op0	op1	op2	Instruction details	Feature
01	00	00	SEL — Four registers	FEAT_SME2
01	00	10	UNALLOCATED	-
01	10	x0	UNALLOCATED	-
11	x0	x0	UNALLOCATED	-
x0	x0	x0	SEL — Two registers	FEAT_SME2
xx	x0	x1	UNALLOCATED	-
xx	x1	xx	UNALLOCATED	-

SME2 Multi-vector - SVE Constructive Binary

SME2 Multi-vector - SVE Constructive Binary

The encodings in this section are decoded from SME encodings.

31	24	23	22	21	20	16	15	13	12	10	9	2	1	0
1 1 0 0 0 0 0 1		op0		1			1 1 0		op1				op2	

op0	op1	op2	Instruction details
00	101	x	SME2 multi-vec quadwords ZIP two registers
01	101	x	UNALLOCATED
10	101	x	UNALLOCATED
11	101	x	SME2 multi-vec saturating shift right narrow two registers
xx	000	x	SME2 multi-vec FCLAMP two registers
xx	001	x	SME2 multi-vec CLAMP two registers
xx	010	0	SME2 multi-vec FCLAMP four registers

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
xx	011	0	SME2 multi-vec CLAMP four registers
xx	01x	1	UNALLOCATED
xx	100	x	SME2 multi-vec ZIP two registers
xx	11x	x	SME2 multi-vec saturating shift right narrow four registers

SME2 multi-vec quadwords ZIP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31							24	23	22	21	20				16	15			13	12			10	9				5	4				1	0
1	1	0	0	0	0	0	1	0	0	1	Zm					1	1	0		1	0	1		Zn					Zd					op

op	Instruction Details	Feature
0	ZIP (two registers)	FEAT_SME2
1	UZP (two registers)	FEAT_SME2

SME2 multi-vec saturating shift right narrow two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31							24	23	22	21	20	19			16	15			13	12			10	9				6	5	4					0
1	1	0	0	0	0	0	1	1	1	1	op	imm4				1	1	0		1	0	1		Zn					U	Zd					

op	U	Instruction Details	Feature
0	0	SQRSHR (two registers)	FEAT_SME2
0	1	UQRSHR (two registers)	FEAT_SME2
1	0	SQRSHRU (two registers)	FEAT_SME2
1	1	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec FCLAMP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31–24	23–22	21	20–16	15–13	12–10	9–5	4–1	0
1 1 0 0 0 0 0 1	size	1	Zm	1 1 0	0 0 0	Zn	Zd	op

size	op	Instruction Details	Feature
xx	1	UNALLOCATED	-
00	0	BFCLAMP	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	0	FCLAMP	FEAT_SME2

SME2 multi-vec CLAMP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31–24	23–22	21	20–16	15–13	12–10	9–5	4–1	0
1 1 0 0 0 0 0 1	size	1	Zm	1 1 0	0 0 1	Zn	Zd	U

U	Instruction Details	Feature
0	SCLAMP	FEAT_SME2
1	UCLAMP	FEAT_SME2

SME2 multi-vec FCLAMP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31–24	23–22	21	20–16	15–13	12–10	9–5	4–2	1	0
1 1 0 0 0 0 0 1	size	1	Zm	1 1 0	0 1 0	Zn	Zd	0	op

size	op	Instruction Details	Feature
xx	1	UNALLOCATED	-
00	0	BFCLAMP	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	0	FCLAMP	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec CLAMP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31							24	23	22	21	20				16	15			13	12			10	9					5	4				2	1	0
1	1	0	0	0	0	0	1	size		1	Zm					1	1	0		0	1	1		Zn						Zd					0	U

U	Instruction Details	Feature
0	SCLAMP	FEAT_SME2
1	UCLAMP	FEAT_SME2

SME2 multi-vec ZIP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31							24	23	22	21	20				16	15			13	12			10	9					5	4					1	0
1	1	0	0	0	0	0	1	size		1	Zm					1	1	0		1	0	0		Zn						Zd						op

op	Instruction Details	Feature
0	ZIP (two registers)	FEAT_SME2
1	UZP (two registers)	FEAT_SME2

SME2 multi-vec saturating shift right narrow four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31							24	23	22	21	20				16	15					11	10	9			7	6	5	4						0
1	1	0	0	0	0	0	1	tsize		1	imm5					1	1	0	1	1		N	Zn				op	U	Zd						

N	op	U	Instruction Details	Feature
x	1	1	UNALLOCATED	-
0	0	0	SQRSHR (four registers)	FEAT_SME2
0	0	1	UQRSHR (four registers)	FEAT_SME2
0	1	0	SQRSHRU (four registers)	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

N	op	U	Instruction Details	Feature
1	0	0	SQRSHRN	FEAT_SME2
1	0	1	UQRSHRN	FEAT_SME2
1	1	0	SQRSHRUN	FEAT_SME2

SME2 Multi-vector - SVE Constructive Unary

SME2 Multi-vector - SVE Constructive Unary

The encodings in this section are decoded from SME encodings.

31							24	23 22	21	20 18	17 16	15					10	9 7	6 5	4 2	1 0
1	1	0	0	0	0	0	1	op0	1	op1	op2	1	1	1	0	0	0		op3		op4

op0	op1	op2	op3	op4	Instruction details
00	000	01	xx	x0	SME2 multi-vec FP to int convert two registers
00	000	10	xx	x0	SME2 multi-vec int to FP two registers
00	100	01	0x	00	SME2 multi-vec FP to int convert four registers
00	100	10	0x	00	SME2 multi-vec int to FP four registers
00	101	00	0x	xx	SME2 multi-vec FP8 down convert four registers
00	101	11	00	x0	SME2 multi-vec quadwords ZIP four registers
01	000	01	xx	x0	UNALLOCATED
01	001	11	xx	xx	UNALLOCATED
01	101	00	0x	1x	UNALLOCATED
01	10x	00	0x	0x	UNALLOCATED
0x	000	00	xx	xx	SME2 multi-vec FP down convert two registers
0x	000	11	xx	xx	SME2 multi-vec int down convert two registers
0x	001	00	x0	xx	SME2 multi-vec FP8 down convert two registers
0x	001	00	x1	xx	UNALLOCATED
10	000	00	xx	xx	SME2 multi-vec convert two registers
10	001	00	1x	xx	UNALLOCATED
10	x01	00	0x	xx	UNALLOCATED
11	00x	00	xx	xx	UNALLOCATED
11	101	00	0x	1x	UNALLOCATED
11	10x	00	0x	0x	UNALLOCATED
1x	000	11	xx	x1	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	Instruction details
1x	000	x1	xx	x0	UNALLOCATED
x0	100	00	0x	0x	UNALLOCATED
xx	000	01	xx	x1	UNALLOCATED
xx	000	10	xx	x1	UNALLOCATED
xx	001	01	xx	xx	SME2 multi-vec unpack two registers
xx	001	10	xx	xx	SME2 multi-vec FP8 up convert two registers
xx	01x	xx	x0	x0	SME2 multi-vec FRINT two registers
xx	100	01	0x	01	UNALLOCATED
xx	100	01	1x	xx	UNALLOCATED
xx	100	10	0x	01	UNALLOCATED
xx	100	11	xx	xx	SME2 multi-vec int down convert four registers
xx	100	!= 11	0x	1x	UNALLOCATED
xx	101	01	x0	0x	SME2 multi-vec unpack four registers
xx	101	01	x0	1x	UNALLOCATED
xx	101	01	x1	xx	UNALLOCATED
xx	101	10	00	x0	SME2 multi-vec ZIP four registers
xx	101	11	1x	xx	UNALLOCATED
xx	101	1x	00	x1	UNALLOCATED
xx	101	1x	01	xx	UNALLOCATED
xx	10x	x0	1x	xx	UNALLOCATED
xx	11x	xx	00	00	SME2 multi-vec FRINT four registers
xx	11x	xx	00	10	UNALLOCATED
xx	11x	xx	10	x0	UNALLOCATED
xx	x1x	xx	x0	x1	UNALLOCATED
xx	x1x	xx	x1	xx	UNALLOCATED
!= 00	000	10	xx	x0	UNALLOCATED
!= 00	100	01	0x	00	UNALLOCATED
!= 00	100	10	0x	00	UNALLOCATED
!= 00	101	11	00	x0	UNALLOCATED
!= 01	001	11	xx	xx	UNALLOCATED

SME2 multi-vec FP to int convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	6	5	4	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1	0	0	0	Zn		U	Zd		0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Feature
0	FCVTZS	FEAT_SME2
1	FCVTZU	FEAT_SME2

SME2 multi-vec int to FP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	6	5	4	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1	1	1	0	0	0	Zn		U	Zd		0

U	Instruction Details	Feature
0	SCVTF	FEAT_SME2
1	UCVTF	FEAT_SME2

SME2 multi-vec FP to int convert four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	Zn		0	U	Zd		0	0

U	Instruction Details	Feature
0	FCVTZS	FEAT_SME2
1	FCVTZU	FEAT_SME2

SME2 multi-vec int to FP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	1	0	1	1	1	0	0	0	Zn		0	U	Zd		0	0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Feature
0	SCVTF	FEAT_SME2
1	UCVTF	FEAT_SME2

SME2 multi-vec FP8 down convert four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	0	Zn		0	N	Zd	

N	Instruction Details	Feature
0	FCVT (narrowing, FP32 to FP8)	FEAT_SME2 && FEAT_FP8
1	FCVTN (FP32 to FP8)	FEAT_SME2 && FEAT_FP8

SME2 multi-vec quadwords ZIP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	Zn		0	0	Zd		op	0

op	Instruction Details	Feature
0	ZIP (four registers)	FEAT_SME2
1	UZP (four registers)	FEAT_SME2

SME2 multi-vec FP down convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31								23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	op	1	0	0	0	0	0	1	1	1	0	0	0	Zn		N	Zd	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	N	Instruction Details	Feature
0	0	FCVT (narrowing, FP32 to FP16)	FEAT_SME2
0	1	FCVTN (FP32 to FP16)	FEAT_SME2
1	0	BFCVT (single-precision to BFloat16)	FEAT_SME2
1	1	BFCVTN	FEAT_SME2

SME2 multi-vec int down convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31								23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	op	1	0	0	0	1	1	1	1	1	0	0	0	Zn		U	Zd	

op	U	Instruction Details	Feature
0	0	SQCVT (two registers)	FEAT_SME2
0	1	UQCVT (two registers)	FEAT_SME2
1	0	SQCVTU (two registers)	FEAT_SME2
1	1	UNALLOCATED	-

SME2 multi-vec FP8 down convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31								23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	op	1	0	0	1	0	0	1	1	1	0	0	0	Zn		0	Zd	

op	Instruction Details	Feature
0	FCVT (narrowing, FP16 to FP8)	FEAT_SME2 && FEAT_FP8
1	BFCVT (BFloat16 to 8-bit floating-point)	FEAT_SME2 && FEAT_FP8

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20			18	17	16	15						10	9			5	4			1	0
1	1	0	0	0	0	0	1	1	0	1	0	0	0		0	0	1	1	1	0	0	0		Zn				Zd				L

L	Instruction Details	Feature
0	FCVT (widening)	FEAT_SME_F16F16
1	FCVTL	FEAT_SME_F16F16

SME2 multi-vec unpack two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20			18	17	16	15						10	9			5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	0	1		0	1	1	1	1	0	0	0		Zn				Zd				U

U	Instruction Details	Feature
0	SUNPK	FEAT_SME2
1	UUNPK	FEAT_SME2

SME2 multi-vec FP8 up convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20			18	17	16	15						10	9			5	4			1	0
1	1	0	0	0	0	0	1	opc		1	0	0	1		1	0	1	1	1	0	0	0		Zn				Zd				L

opc	L	Instruction Details	Feature
00	0	F1CVT, F2CVT — F1CVT	FEAT_SME2 && FEAT_FP8
00	1	F1CVTL, F2CVTL — F1CVTL	FEAT_SME2 && FEAT_FP8
01	0	BF1CVT, BF2CVT — BF1CVT	FEAT_SME2 && FEAT_FP8
01	1	BF1CVTL, BF2CVTL — BF1CVTL	FEAT_SME2 && FEAT_FP8

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	L	Instruction Details	Feature
10	0	F1CVT, F2CVT — F2CVT	FEAT_SME2 && FEAT_FP8
10	1	F1CVTL, F2CVTL — F2CVTL	FEAT_SME2 && FEAT_FP8
11	0	BF1CVT, BF2CVT — BF2CVT	FEAT_SME2 && FEAT_FP8
11	1	BF1CVTL, BF2CVTL — BF2CVTL	FEAT_SME2 && FEAT_FP8

SME2 multi-vec FRINT two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21		19	18		16	15					10	9		6	5	4		1	0
1	1	0	0	0	0	0	1	size		1	0	1	opc			1	1	1	0	0	0	Zn			0	Zd			0

size	opc	Instruction Details	Feature
10	000	FRINTN	FEAT_SME2
10	001	FRINTP	FEAT_SME2
10	010	FRINTM	FEAT_SME2
10	011	UNALLOCATED	-
10	100	FRINTA	FEAT_SME2
10	101	UNALLOCATED	-
10	11x	UNALLOCATED	-
!= 10	xxx	UNALLOCATED	-

SME2 multi-vec int down convert four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9		7	6	5	4		0
1	1	0	0	0	0	0	1	sz	op	1	1	0	0	1	1	1	1	1	0	0	0	Zn			N	U	Zd		

op	N	U	Instruction Details	Feature
0	0	0	SQCVT (four registers)	FEAT_SME2
0	0	1	UQCVT (four registers)	FEAT_SME2
0	1	0	SQCVTN	FEAT_SME2
0	1	1	UQCVTN	FEAT_SME2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	N	U	Instruction Details	Feature
1	x	1	UNALLOCATED	-
1	0	0	SQCVTU (four registers)	FEAT_SME2
1	1	0	SQCVTUN	FEAT_SME2

SME2 multi-vec unpack four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9		6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	1	0	1	0	1	1	1	1	0	0	0	Zn			0	Zd			0	U

U	Instruction Details	Feature
0	SUNPK	FEAT_SME2
1	UUNPK	FEAT_SME2

SME2 multi-vec ZIP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	1	0	1	1	0	1	1	1	0	0	0	Zn			0	0	Zd			op	0

op	Instruction Details	Feature
0	ZIP (four registers)	FEAT_SME2
1	UZP (four registers)	FEAT_SME2

SME2 multi-vec FRINT four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21			19	18		16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	1	1		opc			1	1	1	0	0	0	Zn			0	0	Zd			0	0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Feature
10	000	FRINTN	FEAT_SME2
10	001	FRINTP	FEAT_SME2
10	010	FRINTM	FEAT_SME2
10	011	UNALLOCATED	-
10	100	FRINTA	FEAT_SME2
10	101	UNALLOCATED	-
10	11x	UNALLOCATED	-
!=10	xxx	UNALLOCATED	-

SME2 Multi-vector - FP Multiply

SME2 Multi-vector - FP Multiply

The encodings in this section are decoded from SME encodings.

31							24	23 22	21	20 18	17 16	15					10	9 7	6	5	4 2	1	0
1	1	0	0	0	0	0	1		1		op0	1	1	1	0	0	1		op1	0		op2	0

op0	op1	op2	Instruction details
01	0	0	SME2 multi-vec FP multiply (four registers)
01	0	1	UNALLOCATED
01	1	x	UNALLOCATED
11	x	x	UNALLOCATED
x0	x	x	SME2 multi-vec FP multiply (two registers)

SME2 multi-vec FP multiply (four registers)

The encodings in this section are decoded from SME2 Multi-vector - FP Multiply.

31							24	23 22	21	20 18	17 16	15					10	9 7	6	5	4 2	1	0
1	1	0	0	0	0	0	1	size	1	Zm	0 1	1	1	1	0	0	1	Zn	0	0	Zd	0	0

size	Instruction Details	Feature
00	BFMUL (multiple vectors)	FEAT_SME2 && FEAT_SVE_BFSCALE
!=00	FMUL (multiple vectors)	FEAT_SME2p2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec FP multiply (two registers)

The encodings in this section are decoded from SME2 Multi-vector - FP Multiply.

31							24	23	22	21	20			17	16						10	9			6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	1	1	0	0	1	Zn				0	Zd				0

size	Instruction Details	Feature
00	BFMUL (multiple vectors)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	FMUL (multiple vectors)	FEAT_SME2p2

SME2 Multiple and Single Vector - FP multiply

SME2 Multiple and Single Vector - FP multiply

The encodings in this section are decoded from SME encodings.

31							24	23	22	21	20			17	16	15					10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1			1					op0	1	1	1	0	1	0					op1	0					op2	0

op0	op1	op2	Instruction details
0	x	x	SME2 multiple and single vector FP multiply (two registers)
1	0	0	SME2 multiple and single vector FP multiply (four registers)
1	0	1	UNALLOCATED
1	1	x	UNALLOCATED

SME2 multiple and single vector FP multiply (two registers)

The encodings in this section are decoded from SME2 Multiple and Single Vector - FP multiply.

31							24	23	22	21	20			17	16	15					10	9			6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	1	1	0	1	0	Zn				0	Zd				0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Instruction Details	Feature
00	BFMUL (multiple and single vector)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	FMUL (multiple and single vector)	FEAT_SME2p2

SME2 multiple and single vector FP multiply (four registers)

The encodings in this section are decoded from SME2 Multiple and Single Vector - FP multiply.

31							24	23	22	21	20				17	16	15						10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	Zm					1	1	1	1	0	1	0		Zn				0	0	Zd				0	0

size	Instruction Details	Feature
00	BFMUL (multiple and single vector)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	FMUL (multiple and single vector)	FEAT_SME2p2

SME Memory

SME Memory

The encodings in this section are decoded from SME encodings.

31							25	24				21	20				16	15	14				10	9					5	4			2	1	0
1	1	1	0	0	0	0	op0						op1					op2	op3											op4					

op0	op1	op2	op3	op4	Instruction details	Feature
0xx0	xxxxx	x	xxxxx	0xx	SME load array vector (elements)	-
0xx1	xxxxx	x	xxxxx	0xx	SME store array vector (elements)	-
100x	00000	0	xx000	0xx	SME save and restore array	-
100x	xxxxx	1	00000	000	SME2 lookup table load/store	-
100x	xxxxx	1	00000	001	UNALLOCATED	-
100x	xxxxx	1	00000	01x	UNALLOCATED	-
100x	xxxxx	1	01000	0xx	UNALLOCATED	-
100x	xxxxx	1	1x000	0xx	UNALLOCATED	-
100x	xxxxx	x	xx001	0xx	UNALLOCATED	-
100x	xxxxx	x	xx01x	0xx	UNALLOCATED	-
100x	xxxxx	x	xx1xx	0xx	UNALLOCATED	-
100x	!= 00000	0	xx000	0xx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	Instruction details	Feature
101x	xxxxx	x	xxxxx	0xx	UNALLOCATED	-
110x	xxxxx	x	xxxxx	0xx	UNALLOCATED	-
1110	xxxxx	x	xxxxx	0xx	LD1Q	FEAT_SME
1111	xxxxx	x	xxxxx	0xx	ST1Q	FEAT_SME
xxxx	xxxxx	x	xxxxx	1xx	UNALLOCATED	-

SME load array vector (elements)

The encodings in this section are decoded from SME Memory.

31							24	23	22	21	20			16	15	14	13	12		10	9			5	4	3		0
1	1	1	0	0	0	0	0	msz		0	Rm				V	Rs		Pg			Rn				0	opc		

msz	Instruction Details	Feature
00	LD1B (scalar plus scalar, tile slice)	FEAT_SME
01	LD1H (scalar plus scalar, tile slice)	FEAT_SME
10	LD1W (scalar plus scalar, tile slice)	FEAT_SME
11	LD1D (scalar plus scalar, tile slice)	FEAT_SME

SME store array vector (elements)

The encodings in this section are decoded from SME Memory.

31							24	23	22	21	20			16	15	14	13	12		10	9			5	4	3		0
1	1	1	0	0	0	0	0	msz		1	Rm				V	Rs		Pg			Rn				0	opc		

msz	Instruction Details	Feature
00	ST1B (scalar plus scalar, tile slice)	FEAT_SME
01	ST1H (scalar plus scalar, tile slice)	FEAT_SME
10	ST1W (scalar plus scalar, tile slice)	FEAT_SME
11	ST1D (scalar plus scalar, tile slice)	FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME save and restore array

The encodings in this section are decoded from SME Memory.

31–22	21	20–16	15	14–13	12–10	9–5	4	3–0
1 1 1 0 0 0 0 1 0 0	op	0 0 0 0 0	0	Rv	0 0 0	Rn	0	imm4

op	Instruction Details	Feature
0	LDR (array vector)	FEAT_SME
1	STR (array vector)	FEAT_SME

SME2 lookup table load/store

The encodings in this section are decoded from SME Memory.

31–22	21–16	15	14–10	9–5	4–2	1–0
1 1 1 0 0 0 0 1 0 0	opc	1	0 0 0 0 0	Rn	0 0 0	opc2

opc	opc2	Instruction Details	Feature
x0xxxx	xx	UNALLOCATED	-
x10xxx	xx	UNALLOCATED	-
x110xx	xx	UNALLOCATED	-
x1110x	xx	UNALLOCATED	-
x11110	xx	UNALLOCATED	-
x11111	!= 00	UNALLOCATED	-
011111	00	LDR (table)	FEAT_SME2
111111	00	STR (table)	FEAT_SME2

SVE encodings

SVE encodings

The encodings in this section are decoded from A64 instruction set encoding.

31–29	28–25	24–17	16	15–10	9–5	4	3–0
op0	0 0 1 0	op1		op2		op3	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
000	0xx0xxxx	000xxx	x	SVE Integer Binary Arithmetic - Predicated	-
000	0xx0xxxx	001xxx	x	SVE Integer Reduction	-
000	0xx0xxxx	100xxx	x	SVE Bitwise Shift - Predicated	-
000	0xx0xxxx	101xxx	x	SVE Integer Unary Arithmetic - Predicated	-
000	0xx0xxxx	x1xxxx	x	SVE Integer Multiply-Add - Predicated	-
000	0xx1xxxx	001xxx	x	SVE Bitwise Logical - Unpredicated	-
000	0xx1xxxx	0100xx	x	SVE Index Generation	-
000	0xx1xxxx	0101xx	x	SVE Stack Allocation	-
000	0xx1xxxx	011xxx	x	SVE2 Integer Arithmetic - Unpredicated	-
000	0xx1xxxx	100xxx	x	SVE Bitwise Shift - Unpredicated	-
000	0xx1xxxx	1011xx	x	SVE Integer Misc - Unpredicated	-
000	0xx1xxxx	11xxxx	x	SVE Element Count	-
000	10x1xxxx	000xxx	x	SVE Permute Vector - Extract	-
000	11x1xxxx	000xxx	x	SVE Permute Vector - Segments	-
000	1xx00xxx	xxxxxx	x	SVE Bitwise Immediate	-
000	1xx01xxx	xxxxxx	x	SVE Integer Wide Immediate - Predicated	-
000	1xx1xxxx	001001	x	SVE Permute Vector - One Source Quadwords	-
000	1xx1xxxx	001110	x	SVE Permute Vector - Unpredicated	-
000	1xx1xxxx	001111	x	UNALLOCATED	-
000	1xx1xxxx	010xxx	x	SVE Permute Predicate	-
000	1xx1xxxx	10xxxx	x	SVE Permute Vector - Predicated	-
001	0xx0xxxx	xxxxxx	x	SVE Integer Compare - Vectors	-
001	1xx00xxx	11xxxx	x	SVE Propagate Break	-
001	1xx01xxx	01xxxx	x	SVE Partition Break	-
001	1xx01xxx	11xxxx	x	SVE Predicate Misc	-
001	1xx100xx	10xxxx	x	SVE Predicate Count	-
001	1xx101xx	1000xx	x	SVE Inc/Dec by Predicate Count	-
001	1xx101xx	1001xx	x	SVE Write FFR	-
001	1xx101xx	101xxx	x	UNALLOCATED	-
001	1xx11xxx	10xxxx	x	UNALLOCATED	-
001	1xx1xxxx	00xxxx	x	SVE Integer Compare - Scalars	-
001	1xx1xxxx	01xxxx	1	SVE Scalar Integer Compare - Predicate-as-counter	-
001	1xx1xxxx	11xxxx	x	SVE Integer Wide Immediate - Unpredicated	-
010	0x10xxxx	11001x	x	UNALLOCATED	-
010	0xx0xxxx	0xxxxx	x	SVE Integer Multiply-Add - Unpredicated	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
010	0xx0xxxx	10xxxx	x	SVE2 Integer - Predicated	-
010	0xx0xxxx	1101x1	x	UNALLOCATED	-
010	0xx1xxxx	xxxxxx	x	SVE Multiply - Indexed	-
010	1xx0xxxx	0xxxxx	x	SVE2 Widening Integer Arithmetic	-
010	1xx0xxxx	10xxxx	x	SVE Misc	-
010	1xx0xxxx	11xxxx	x	SVE2 Accumulate	-
010	1xx1xxxx	0xxxxx	x	SVE2 Narrowing	-
010	1xx1xxxx	101xxx	x	SVE2 Histogram Computation (Segment) and Lookup Table	-
010	1xx1xxxx	111xxx	x	SVE2 Crypto Extensions	-
011	01x1xxxx	111000	x	UNALLOCATED	-
011	0x01xxxx	0111xx	x	UNALLOCATED	-
011	0x01xxxx	10xx10	x	SVE2 FP8 widening multiply-add	-
011	0x01xxxx	10xx11	x	UNALLOCATED	-
011	0x01xxxx	11x1xx	x	UNALLOCATED	-
011	0x11xxxx	10xx1x	x	UNALLOCATED	-
011	0x11xxxx	x1x1xx	x	UNALLOCATED	-
011	0xx00001	100xxx	x	UNALLOCATED	-
011	0xx0010x	100xxx	x	UNALLOCATED	-
011	0xx00x0x	11xxxx	x	UNALLOCATED	-
011	0xx00x1x	1xxxxx	x	UNALLOCATED	-
011	0xx010xx	11xxxx	x	UNALLOCATED	-
011	0xx011xx	1xxxxx	x	SVE2 floating-point unary operations - zeroing predicated	-
011	0xx1xxxx	001011	x	UNALLOCATED	-
011	0xx1xxxx	0011xx	x	UNALLOCATED	-
011	0xx1xxxx	01x0xx	x	SVE floating-point widening multiply-add - indexed	-
011	0xx1xxxx	10x00x	x	SVE floating-point widening multiply-add	-
011	0xx1xxxx	10x10x	x	UNALLOCATED	-
011	0xx1xxxx	11101x	x	UNALLOCATED	-
011	1xx001xx	0010xx	x	UNALLOCATED	-
011	1xx001xx	0011xx	x	SVE floating-point unary operations - unpredicated	-
011	1xx010xx	001xxx	x	SVE floating-point compare - with zero	-
011	1xx011xx	001xxx	x	SVE floating-point accumulating reduction	-
011	1xx0xxxx	100xxx	x	SVE floating-point arithmetic - predicated	-
011	1xx0xxxx	101xxx	x	SVE floating-point unary operations - merging predicated	-
011	1xx1xxxx	xxxxxx	x	SVE floating-point multiply-add	-
100	xxxxxxxx	xxxxxx	x	SVE Memory - 32-bit Gather and Unsized Contiguous	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
101	xxxxxxxx	xxxxxx	x	SVE Memory - Contiguous Load	-
110	xxxxxxxx	xxxxxx	x	SVE Memory - 64-bit Gather	-
111	xxxxxxxx	001xxx	x	SVE Memory - Non-temporal and Quadword Scatter Store	-
111	xxxxxxxx	011xxx	x	SVE Memory - Non-temporal and Multi-register Contiguous Store	-
111	xxxxxxxx	0x0xxx	x	SVE Memory - Contiguous Store and Unsized Contiguous	-
111	xxxxxxxx	101xxx	x	SVE Memory - Scatter	-
111	xxxxxxxx	111xxx	x	SVE Memory - Contiguous Store with Immediate Offset	-
111	xxxxxxxx	1x0xxx	x	SVE Memory - Scatter with Optional Sign Extend	-
000	0xx1xxxx	000xxx	x	SVE integer add/subtract vectors (unpredicated)	-
000	0xx1xxxx	1010xx	x	SVE address generation	-
000	1xx1xxxx	001000	x	DUP (indexed)	FEAT_SVE || FEAT_SME
000	1xx1xxxx	00101x	x	SVE table lookup (three sources)	-
000	1xx1xxxx	001100	x	TBL — Single register table	FEAT_SVE || FEAT_SME
000	1xx1xxxx	001101	x	TBXQ	FEAT_SVE2p1 || FEAT_SME2p1
000	1xx1xxxx	011xxx	x	SVE permute vector elements	-
000	1xx1xxxx	11xxxx	x	SEL (vectors)	FEAT_SVE || FEAT_SME
001	0xx1xxxx	xxxxxx	x	SVE integer compare with unsigned immediate	-
001	1xx00xxx	01xxxx	x	SVE predicate logical operations	-
001	1xx0xxxx	x0xxxx	x	SVE integer compare with signed immediate	-
001	1xx1xxxx	01xxxx	0	SVE broadcast predicate element	-
010	0000xxxx	11001x	x	SVE two-way dot product	-
010	0100xxxx	11001x	x	SVE two-way dot product (indexed)	-
010	0xx0xxxx	11000x	x	SVE integer clamp	-
010	0xx0xxxx	1101x0	x	SVE2 multiply-add (checked pointer)	-
010	0xx0xxxx	111xxx	x	SVE permute vector elements (quadwords)	-
010	1xx1xxxx	100xxx	x	SVE2 character match	-
010	1xx1xxxx	110xxx	x	HISTCNT	FEAT_SVE2
011	00x1xxxx	111000	x	SVE2 FP8 matrix multiply-accumulate	-
011	0x01xxxx	0101xx	x	SVE2 FP8 multiply-add long (indexed)	-
011	0xx00000	100xxx	x	FCADD	FEAT_SVE || FEAT_SME
011	0xx0000x	101xxx	x	SVE floating-point convert (top, predicated)	-
011	0xx0010x	101xxx	x	SVE floating-point convert precision odd elements	-
011	0xx010xx	100xxx	x	SVE2 floating-point pairwise operations	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
011	0xx010xx	101xxx	x	SVE floating-point recursive reduction (quadwords)	-
011	0xx0xxxx	0xxxxx	x	FCMLA (vectors)	FEAT_SVE || FEAT_SME
011	0xx1xxxx	0000xx	x	SVE floating-point multiply-add (indexed)	-
011	0xx1xxxx	0001xx	x	SVE floating-point complex multiply-add (indexed)	-
011	0xx1xxxx	001001	x	SVE FP clamp	-
011	0xx1xxxx	0010x0	x	SVE floating-point multiply (indexed)	-
011	0xx1xxxx	1100xx	x	SVE2 FP8 multiply-add long long (indexed)	-
011	0xx1xxxx	111001	x	SVE floating-point matrix multiply accumulate	-
011	1xx000xx	001xxx	x	SVE floating-point recursive reduction	-
011	1xx0xxxx	000xxx	x	SVE floating-point arithmetic (unpredicated)	-
011	1xx0xxxx	x1xxxx	x	SVE floating-point compare vectors	-

SVE Integer Binary Arithmetic - Predicated

SVE Integer Binary Arithmetic - Predicated

The encodings in this section are decoded from SVE encodings.

31	24	23	22	21	20	18	17	16	15	13	12	0
0 0 0 0 0 1 0 0				0	op0				0 0 0			

op0	Instruction details
00x	SVE integer add/subtract vectors (predicated)
01x	SVE integer min/max/difference (predicated)
100	SVE integer multiply vectors (predicated)
101	SVE integer divide vectors (predicated)
11x	SVE bitwise logical operations (predicated)

SVE integer add/subtract vectors (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31	24	23	22	21	19	18	16	15	13	12	10	9	5	4	0
0 0 0 0 0 1 0 0		size		0 0 0		opc		0 0 0		Pg		Zm		Zdn	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Feature
xx	000	ADD (vectors, predicated)	FEAT_SVE || FEAT_SME
xx	001	SUB (vectors, predicated)	FEAT_SVE || FEAT_SME
xx	010	UNALLOCATED	-
xx	011	SUBR (vectors)	FEAT_SVE || FEAT_SME
11	100	ADDPT (predicated)	FEAT_SVE && FEAT_CPA
11	101	SUBPT (predicated)	FEAT_SVE && FEAT_CPA
11	11x	UNALLOCATED	-
!= 11	1xx	UNALLOCATED	-

SVE integer min/max/difference (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0	size		0	0	1	opc		U	0	0	0	Pg			Zm			Zdn		

opc	U	Instruction Details	Feature
00	0	SMAX (vectors)	FEAT_SVE || FEAT_SME
00	1	UMAX (vectors)	FEAT_SVE || FEAT_SME
01	0	SMIN (vectors)	FEAT_SVE || FEAT_SME
01	1	UMIN (vectors)	FEAT_SVE || FEAT_SME
10	0	SABD	FEAT_SVE || FEAT_SME
10	1	UABD	FEAT_SVE || FEAT_SME
11	x	UNALLOCATED	-

SVE integer multiply vectors (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21	20		18	17	16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0	size		0	1	0	0	H	U	0	0	0	Pg			Zm			Zdn		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H	U	Instruction Details	Feature
0	0	MUL (vectors, predicated)	FEAT_SVE || FEAT_SME
0	1	UNALLOCATED	-
1	0	SMULH (predicated)	FEAT_SVE || FEAT_SME
1	1	UMULH (predicated)	FEAT_SVE || FEAT_SME

SVE integer divide vectors (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21	20		18	17	16	15			13	12		10	9		5	4		0
0	0	0	0	0	1	0	0	size		0	1	0	1	R	U	0	0	0		Pg			Zm			Zdn		

R	U	Instruction Details	Feature
0	0	SDIV	FEAT_SVE || FEAT_SME
0	1	UDIV	FEAT_SVE || FEAT_SME
1	0	SDIVR	FEAT_SVE || FEAT_SME
1	1	UDIVR	FEAT_SVE || FEAT_SME

SVE bitwise logical operations (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21		19	18		16	15			13	12		10	9		5	4		0
0	0	0	0	0	1	0	0	size		0	1	1	opc			0	0	0		Pg			Zm			Zdn		

opc	Instruction Details	Feature
000	ORR (vectors, predicated)	FEAT_SVE || FEAT_SME
001	EOR (vectors, predicated)	FEAT_SVE || FEAT_SME
010	AND (vectors, predicated)	FEAT_SVE || FEAT_SME
011	BIC (vectors, predicated)	FEAT_SVE || FEAT_SME
1xx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE Integer Reduction

SVE Integer Reduction

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20		18	17	16	15			13	12											0
0	0	0	0	0	1	0	0			0	op0					0	0	1													

op0	Instruction details
000	SVE integer add reduction (predicated)
001	SVE integer add reduction (quadwords)
010	SVE integer min/max reduction (predicated)
011	SVE integer min/max reduction (quadwords)
10x	SVE constructive prefix (predicated)
110	SVE bitwise logical reduction (predicated)
111	SVE bitwise logical reduction (quadwords)

SVE integer add reduction (predicated)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21	20			18	17	16	15			13	12			10	9					5	4					0
0	0	0	0	0	1	0	0			0	0	0	0		op	U	0	0	1		Pg				Zn						Vd					

op	U	Instruction Details	Feature
0	0	SADDV	FEAT_SVE || FEAT_SME
0	1	UADDV	FEAT_SVE || FEAT_SME
1	x	UNALLOCATED	-

SVE integer add reduction (quadwords)

The encodings in this section are decoded from SVE Integer Reduction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20		18	17	16	15			13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	0	0	1	op	U	0	0	1		Pg			Zn			Vd		

op	U	Instruction Details	Feature
0	0	UNALLOCATED	-
0	1	ADDQV	FEAT_SVE2p1 || FEAT_SME2p1
1	x	UNALLOCATED	-

SVE integer min/max reduction (predicated)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21	20		18	17	16	15			13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	0	1	0	op	U	0	0	1		Pg			Zn			Vd		

op	U	Instruction Details	Feature
0	0	SMAXV	FEAT_SVE || FEAT_SME
0	1	UMAXV	FEAT_SVE || FEAT_SME
1	0	SMINV	FEAT_SVE || FEAT_SME
1	1	UMINV	FEAT_SVE || FEAT_SME

SVE integer min/max reduction (quadwords)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21	20		18	17	16	15			13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	0	1	1	op	U	0	0	1		Pg			Zn			Vd		

op	U	Instruction Details	Feature
0	0	SMAXQV	FEAT_SVE2p1 || FEAT_SME2p1
0	1	UMAXQV	FEAT_SVE2p1 || FEAT_SME2p1
1	0	SMINQV	FEAT_SVE2p1 || FEAT_SME2p1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	Instruction Details	Feature
1	1	UMINQV	FEAT_SVE2p1 || FEAT_SME2p1

SVE constructive prefix (predicated)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	1	0	opc		M	0	0	1	Pg			Zn			Zd		

opc	Instruction Details	Feature
00	MOVPRFX (predicated)	FEAT_SVE || FEAT_SME
!= 00	UNALLOCATED	-

SVE bitwise logical reduction (predicated)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21	20		18	17	16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	1	1	0	opc		0	0	1	Pg			Zn			Vd		

opc	Instruction Details	Feature
00	ORV	FEAT_SVE || FEAT_SME
01	EORV	FEAT_SVE || FEAT_SME
10	ANDV	FEAT_SVE || FEAT_SME
11	UNALLOCATED	-

SVE bitwise logical reduction (quadwords)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21	20		18	17	16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	1	1	1	opc		0	0	1	Pg			Zn			Vd		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
00	ORQV	FEAT_SVE2p1 || FEAT_SME2p1
01	EORQV	FEAT_SVE2p1 || FEAT_SME2p1
10	ANDQV	FEAT_SVE2p1 || FEAT_SME2p1
11	UNALLOCATED	-

SVE Bitwise Shift - Predicated

SVE Bitwise Shift - Predicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19	18		16	15		13	12													0
0	0	0	0	0	1	0	0			0	op0					1	0	0														

op0	Instruction details
0x	SVE bitwise shift by immediate (predicated)
10	SVE bitwise shift by vector (predicated)
11	SVE bitwise shift by wide elements (predicated)

SVE bitwise shift by immediate (predicated)

The encodings in this section are decoded from SVE Bitwise Shift - Predicated.

31							24	23	22	21	20	19	18	17	16	15		13	12		10	9	8	7		5	4					0
0	0	0	0	0	1	0	0	tszh		0	0	opc		L	U	1	0	0	Pg			tszl		imm3			Zdn					

opc	L	U	Instruction Details	Feature
00	0	0	ASR (immediate, predicated)	FEAT_SVE || FEAT_SME
00	0	1	LSR (immediate, predicated)	FEAT_SVE || FEAT_SME
00	1	0	UNALLOCATED	-
00	1	1	LSL (immediate, predicated)	FEAT_SVE || FEAT_SME
01	0	0	ASRD	FEAT_SVE || FEAT_SME
01	0	1	UNALLOCATED	-
01	1	0	SQSHL (immediate)	FEAT_SVE2 || FEAT_SME
01	1	1	UQSHL (immediate)	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	L	U	Instruction Details	Feature
10	x	x	UNALLOCATED	-
11	0	0	SRSHR	FEAT_SVE2 || FEAT_SME
11	0	1	URSHR	FEAT_SVE2 || FEAT_SME
11	1	0	UNALLOCATED	-
11	1	1	SQSHLU	FEAT_SVE2 || FEAT_SME

SVE bitwise shift by vector (predicated)

The encodings in this section are decoded from SVE Bitwise Shift - Predicated.

31							24	23	22	21	20	19	18	17	16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	1	0	R	L	U	1	0	0	Pg			Zm			Zdn		

R	L	U	Instruction Details	Feature
x	1	0	UNALLOCATED	-
0	0	0	ASR (vectors)	FEAT_SVE || FEAT_SME
0	0	1	LSR (vectors)	FEAT_SVE || FEAT_SME
0	1	1	LSL (vectors)	FEAT_SVE || FEAT_SME
1	0	0	ASRR	FEAT_SVE || FEAT_SME
1	0	1	LSRR	FEAT_SVE || FEAT_SME
1	1	1	LSLR	FEAT_SVE || FEAT_SME

SVE bitwise shift by wide elements (predicated)

The encodings in this section are decoded from SVE Bitwise Shift - Predicated.

31							24	23	22	21	20	19	18	17	16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	1	1	R	L	U	1	0	0	Pg			Zm			Zdn		

R	L	U	Instruction Details	Feature
0	0	0	ASR (wide elements, predicated)	FEAT_SVE || FEAT_SME
0	0	1	LSR (wide elements, predicated)	FEAT_SVE || FEAT_SME
0	1	0	UNALLOCATED	-
0	1	1	LSL (wide elements, predicated)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R	L	U	Instruction Details	Feature
1	x	x	UNALLOCATED	-

SVE Integer Unary Arithmetic - Predicated

SVE Integer Unary Arithmetic - Predicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19	18			16	15			13	12																0
0	0	0	0	0	1	0	0			0	op0						1	0	1																		

op0	Instruction details
x0	SVE integer unary operations (predicated)
x1	SVE bitwise unary operations (predicated)

SVE integer unary operations (predicated)

The encodings in this section are decoded from SVE Integer Unary Arithmetic - Predicated.

31							24	23	22	21	20	19	18			16	15			13	12			10	9					5	4						0
0	0	0	0	0	1	0	0	size		0	M	0	opc				1	0	1		Pg				Zn						Zd						

M	opc	Instruction Details	Feature
0	000	SXTB, SXTH, SXTW — Byte, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	001	UXTB, UXTH, UXTW — Byte, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	010	SXTB, SXTH, SXTW — Halfword, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	011	UXTB, UXTH, UXTW — Halfword, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	100	SXTB, SXTH, SXTW — Word, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	101	UXTB, UXTH, UXTW — Word, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	110	ABS — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	111	NEG — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	000	SXTB, SXTH, SXTW — Byte, merging	FEAT_SVE || FEAT_SME
1	001	UXTB, UXTH, UXTW — Byte, merging	FEAT_SVE || FEAT_SME
1	010	SXTB, SXTH, SXTW — Halfword, merging	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	opc	Instruction Details	Feature
1	011	UXTB, UXTH, UXTW — Halfword, merging	FEAT_SVE || FEAT_SME
1	100	SXTB, SXTH, SXTW — Word, merging	FEAT_SVE || FEAT_SME
1	101	UXTB, UXTH, UXTW — Word, merging	FEAT_SVE || FEAT_SME
1	110	ABS — Merging	FEAT_SVE || FEAT_SME
1	111	NEG — Merging	FEAT_SVE || FEAT_SME

SVE bitwise unary operations (predicated)

The encodings in this section are decoded from SVE Integer Unary Arithmetic - Predicated.

31							24	23 22	21	20	19	18 16	15 13	12 10	9 5	4 0
0	0	0	0	0	1	0	0	size	0	M	1	opc	1 0 1	Pg	Zn	Zd

M	opc	Instruction Details	Feature
x	111	UNALLOCATED	-
0	000	CLS — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	001	CLZ — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	010	CNT — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	011	CNOT — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	100	FABS — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	101	FNEG — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	110	NOT (vector) — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	000	CLS — Merging	FEAT_SVE || FEAT_SME
1	001	CLZ — Merging	FEAT_SVE || FEAT_SME
1	010	CNT — Merging	FEAT_SVE || FEAT_SME
1	011	CNOT — Merging	FEAT_SVE || FEAT_SME
1	100	FABS — Merging	FEAT_SVE || FEAT_SME
1	101	FNEG — Merging	FEAT_SVE || FEAT_SME
1	110	NOT (vector) — Merging	FEAT_SVE || FEAT_SME

SVE Integer Multiply-Add - Predicated

SVE Integer Multiply-Add - Predicated

The encodings in this section are decoded from SVE encodings.

31							25	24	23 22	21	20 16	15	14	13 0
0	0	0	0	0	1	0		0		0		op0	1	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0	SVE integer multiply-accumulate writing addend (predicated)
1	SVE integer multiply-add writing multiplicand (predicated)

SVE integer multiply-accumulate writing addend (predicated)

The encodings in this section are decoded from SVE Integer Multiply-Add - Predicated.

31–24	23–22	21	20–16	15	14	13	12–10	9–5	4–0
0 0 0 0 0 1 0 0	size	0	Zm	0	1	op	Pg	Zn	Zda

op	Instruction Details	Feature
0	MLA (vectors)	FEAT_SVE || FEAT_SME
1	MLS (vectors)	FEAT_SVE || FEAT_SME

SVE integer multiply-add writing multiplicand (predicated)

The encodings in this section are decoded from SVE Integer Multiply-Add - Predicated.

31–24	23–22	21	20–16	15	14	13	12–10	9–5	4–0
0 0 0 0 0 1 0 0	size	0	Zm	1	1	op	Pg	Za	Zdn

op	Instruction Details	Feature
0	MAD	FEAT_SVE || FEAT_SME
1	MSB	FEAT_SVE || FEAT_SME

SVE Bitwise Logical - Unpredicated

SVE Bitwise Logical - Unpredicated

The encodings in this section are decoded from SVE encodings.

31–25	24	23–22	21	20–16	15–13	12–10	9–0
0 0 0 0 0 1 0	0		1		0 0 1	op0	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details	Feature
0xx	UNALLOCATED	-
100	SVE bitwise logical operations (unpredicated)	-
101	XAR	FEAT_SVE2 || FEAT_SME
11x	SVE2 bitwise ternary operations	-

SVE bitwise logical operations (unpredicated)

The encodings in this section are decoded from SVE Bitwise Logical - Unpredicated.

31							24	23	22	21	20				16	15			13	12			10	9				5	4				0
0	0	0	0	0	1	0	0	opc		1	Zm					0	0	1		1	0	0		Zn					Zd				

opc	Instruction Details	Feature
00	AND (vectors, unpredicated)	FEAT_SVE || FEAT_SME
01	ORR (vectors, unpredicated)	FEAT_SVE || FEAT_SME
10	EOR (vectors, unpredicated)	FEAT_SVE || FEAT_SME
11	BIC (vectors, unpredicated)	FEAT_SVE || FEAT_SME

SVE2 bitwise ternary operations

The encodings in this section are decoded from SVE Bitwise Logical - Unpredicated.

31							24	23	22	21	20				16	15					11	10	9				5	4				0
0	0	0	0	0	1	0	0	opc		1	Zm					0	0	1	1	1		o2	Zk					Zdn				

opc	o2	Instruction Details	Feature
00	0	EOR3	FEAT_SVE2 || FEAT_SME
00	1	BSL	FEAT_SVE2 || FEAT_SME
01	0	BCAX	FEAT_SVE2 || FEAT_SME
01	1	BSL1N	FEAT_SVE2 || FEAT_SME
1x	0	UNALLOCATED	-
10	1	BSL2N	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	Instruction Details	Feature
11	1	NBSL	FEAT_SVE2 || FEAT_SME

SVE Index Generation

SVE Index Generation

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20				16	15			12	11	10	9							0
0	0	0	0	0	1	0	0			1						0	1	0	0	op0									

op0	Instruction details	Feature
00	INDEX (immediates)	FEAT_SVE || FEAT_SME
01	INDEX (scalar, immediate)	FEAT_SVE || FEAT_SME
10	INDEX (immediate, scalar)	FEAT_SVE || FEAT_SME
11	INDEX (scalars)	FEAT_SVE || FEAT_SME

SVE Stack Allocation

SVE Stack Allocation

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20				16	15			12	11	10								0
0	0	0	0	0	1	0	0	op0		1						0	1	0	1	op1									

op0	op1	Instruction details
0	0	SVE stack frame adjustment
0	1	Streaming SVE stack frame adjustment
1	0	SVE stack frame size
1	1	Streaming SVE stack frame size

SVE stack frame adjustment

The encodings in this section are decoded from SVE Stack Allocation.

31							24	23	22	21	20				16	15			12	11	10				5	4			0
0	0	0	0	0	1	0	0	0	op	1	Rn					0	1	0	1	0	imm6					Rd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Feature
0	ADDVL	FEAT_SVE || FEAT_SME
1	ADDPL	FEAT_SVE || FEAT_SME

Streaming SVE stack frame adjustment

The encodings in this section are decoded from SVE Stack Allocation.

31							24	23	22	21	20				16	15			12	11	10				5	4			0
0	0	0	0	0	1	0	0	0	op	1	Rn					0	1	0	1	1	imm6					Rd			

op	Instruction Details	Feature
0	ADDSVL	FEAT_SME
1	ADDSPL	FEAT_SME

SVE stack frame size

The encodings in this section are decoded from SVE Stack Allocation.

31							24	23	22	21	20				16	15			12	11	10				5	4			0
0	0	0	0	0	1	0	0	1	op	1	opc2					0	1	0	1	0	imm6					Rd			

op	opc2	Instruction Details	Feature
0	11111	RDVL	FEAT_SVE || FEAT_SME
0	!= 11111	UNALLOCATED	-
1	xxxxx	UNALLOCATED	-

Streaming SVE stack frame size

The encodings in this section are decoded from SVE Stack Allocation.

31							24	23	22	21	20				16	15			12	11	10				5	4			0
0	0	0	0	0	1	0	0	1	op	1	opc2					0	1	0	1	1	imm6					Rd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	opc2	Instruction Details	Feature
0	11111	RDSVL	FEAT_SME
0	!= 11111	UNALLOCATED	-
1	xxxxx	UNALLOCATED	-

SVE2 Integer Arithmetic - Unpredicated

SVE2 Integer Arithmetic - Unpredicated

The encodings in this section are decoded from SVE encodings.

31	25	24	23	22	21	20	16	15	13	12	10	9	0
0 0 0 0 0 1 0		0			1			0 1 1		op0			

op0	Instruction details
0xx	SVE2 integer multiply vectors (unpredicated)
10x	SVE2 signed saturating doubling multiply high (unpredicated)
11x	UNALLOCATED

SVE2 integer multiply vectors (unpredicated)

The encodings in this section are decoded from SVE2 Integer Arithmetic - Unpredicated.

31	24	23	22	21	20	16	15	12	11	10	9	5	4	0
0 0 0 0 0 1 0	0	size		1	Zm		0 1 1	0	opc		Zn		Zd	

size	opc	Instruction Details	Feature
xx	00	MUL (vectors, unpredicated)	FEAT_SVE2 || FEAT_SME
xx	10	SMULH (unpredicated)	FEAT_SVE2 || FEAT_SME
xx	11	UMULH (unpredicated)	FEAT_SVE2 || FEAT_SME
00	01	PMUL	FEAT_SVE2 || FEAT_SME
!= 00	01	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 signed saturating doubling multiply high (unpredicated)

The encodings in this section are decoded from SVE2 Integer Arithmetic - Unpredicated.

31–24	23–22	21	20–16	15–11	10	9–5	4–0
0 0 0 0 0 1 0 0	size	1	Zm	0 1 1 1 0	R	Zn	Zd

R	Instruction Details	Feature
0	SQDMULH (vectors)	FEAT_SVE2 || FEAT_SME
1	SQRDMULH (vectors)	FEAT_SVE2 || FEAT_SME

SVE Bitwise Shift - Unpredicated

SVE Bitwise Shift - Unpredicated

The encodings in this section are decoded from SVE encodings.

31–25	24	23–22	21	20–16	15–13	12	11–0
0 0 0 0 0 1 0	0		1		1 0 0	op0	

op0	Instruction details
0	SVE bitwise shift by wide elements (unpredicated)
1	SVE bitwise shift by immediate (unpredicated)

SVE bitwise shift by wide elements (unpredicated)

The encodings in this section are decoded from SVE Bitwise Shift - Unpredicated.

31–24	23–22	21	20–16	15–13	12	11–10	9–5	4–0
0 0 0 0 0 1 0 0		1	Zm	1 0 0	0	opc	Zn	Zd

opc	Instruction Details	Feature
00	ASR (wide elements, unpredicated)	FEAT_SVE || FEAT_SME
01	LSR (wide elements, unpredicated)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
10	UNALLOCATED	-
11	LSL (wide elements, unpredicated)	FEAT_SVE || FEAT_SME

SVE bitwise shift by immediate (unpredicated)

The encodings in this section are decoded from SVE Bitwise Shift - Unpredicated.

31							24	23	22	21	20	19	18		16	15		13	12	11	10	9		5	4		0
0	0	0	0	0	1	0	0	tszh		1	tszl		imm3			1	0	0	1	opc		Zn			Zd		

opc	Instruction Details	Feature
00	ASR (immediate, unpredicated)	FEAT_SVE || FEAT_SME
01	LSR (immediate, unpredicated)	FEAT_SVE || FEAT_SME
10	UNALLOCATED	-
11	LSL (immediate, unpredicated)	FEAT_SVE || FEAT_SME

SVE Integer Misc - Unpredicated

SVE Integer Misc - Unpredicated

The encodings in this section are decoded from SVE encodings.

31							25	24	23	22	21	20		16	15			12	11	10	9		0
0	0	0	0	0	1	0		0			1				1	0	1	1	op0				

op0	Instruction details
0x	SVE floating-point trig select coefficient
10	SVE floating-point exponential accelerator
11	SVE constructive prefix (unpredicated)

SVE floating-point trig select coefficient

The encodings in this section are decoded from SVE Integer Misc - Unpredicated.

31							24	23	22	21	20		16	15				11	10	9		5	4		0
0	0	0	0	0	1	0	0			1		Zm		1	0	1	1	0	op		Zn			Zd	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Feature
0	FTSSEL	FEAT_SVE
1	UNALLOCATED	-

SVE floating-point exponential accelerator

The encodings in this section are decoded from SVE Integer Misc - Unpredicated.

31	24	23	22	21	20	16	15	12	11	10	9	5	4	0
0 0 0 0 0 1 0 0				1	opc		1 0 1 1		1	0	Zn		Zd	

opc	Instruction Details	Feature
00000	FEXPA	FEAT_SVE || FEAT_SSVE_FEXPA
!= 00000	UNALLOCATED	-

SVE constructive prefix (unpredicated)

The encodings in this section are decoded from SVE Integer Misc - Unpredicated.

31	24	23	22	21	20	16	15	12	11	10	9	5	4	0
0 0 0 0 0 1 0 0		opc		1	opc2		1 0 1 1		1	1	Zn		Zd	

opc	opc2	Instruction Details	Feature
00	00000	MOVPRFX (unpredicated)	FEAT_SVE || FEAT_SME
00	!= 00000	UNALLOCATED	-
!= 00	xxxxx	UNALLOCATED	-

SVE Element Count

SVE Element Count

The encodings in this section are decoded from SVE encodings.

31	24	23	22	21	20	19	16	15	14	13	11	10	0
0 0 0 0 0 1 0 0				1	op0			1	1	op1			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details
0	00x	SVE saturating inc/dec vector by element count
0	100	SVE element count
0	101	UNALLOCATED
1	000	SVE inc/dec vector by element count
1	100	SVE inc/dec register by element count
1	x01	UNALLOCATED
x	01x	UNALLOCATED
x	11x	SVE saturating inc/dec register by element count

SVE saturating inc/dec vector by element count

The encodings in this section are decoded from SVE Element Count.

31							24	23 22	21	20	19 16	15			12	11	10	9 5	4 0
0	0	0	0	0	1	0	0	size	1	0	imm4	1	1	0	0	D	U	pattern	Zdn

size	D	U	Instruction Details	Feature
00	x	x	UNALLOCATED	-
01	0	0	SQINCH (vector)	FEAT_SVE || FEAT_SME
01	0	1	UQINCH (vector)	FEAT_SVE || FEAT_SME
01	1	0	SQDECH (vector)	FEAT_SVE || FEAT_SME
01	1	1	UQDECH (vector)	FEAT_SVE || FEAT_SME
10	0	0	SQINCW (vector)	FEAT_SVE || FEAT_SME
10	0	1	UQINCW (vector)	FEAT_SVE || FEAT_SME
10	1	0	SQDECW (vector)	FEAT_SVE || FEAT_SME
10	1	1	UQDECW (vector)	FEAT_SVE || FEAT_SME
11	0	0	SQINCD (vector)	FEAT_SVE || FEAT_SME
11	0	1	UQINCD (vector)	FEAT_SVE || FEAT_SME
11	1	0	SQDECD (vector)	FEAT_SVE || FEAT_SME
11	1	1	UQDECD (vector)	FEAT_SVE || FEAT_SME

SVE element count

The encodings in this section are decoded from SVE Element Count.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20	19			16	15	14	13			11	10	9				5	4				0
0	0	0	0	0	1	0	0	size		1	0	imm4				1	1	1	0	0		op	pattern					Rd				

size	op	Instruction Details	Feature
xx	1	UNALLOCATED	-
00	0	CNTB, CNTD, CNTH, CNTW — Byte	FEAT_SVE || FEAT_SME
01	0	CNTB, CNTD, CNTH, CNTW — Halfword	FEAT_SVE || FEAT_SME
10	0	CNTB, CNTD, CNTH, CNTW — Word	FEAT_SVE || FEAT_SME
11	0	CNTB, CNTD, CNTH, CNTW — Doubleword	FEAT_SVE || FEAT_SME

SVE inc/dec vector by element count

The encodings in this section are decoded from SVE Element Count.

31							24	23	22	21	20	19			16	15	14	13			11	10	9				5	4				0
0	0	0	0	0	1	0	0	size		1	1	imm4				1	1	0	0	0		D	pattern					Zdn				

size	D	Instruction Details	Feature
00	x	UNALLOCATED	-
01	0	INCD, INCH, INCW (vector) — Halfword	FEAT_SVE || FEAT_SME
01	1	DECD, DECH, DECW (vector) — Halfword	FEAT_SVE || FEAT_SME
10	0	INCD, INCH, INCW (vector) — Word	FEAT_SVE || FEAT_SME
10	1	DECD, DECH, DECW (vector) — Word	FEAT_SVE || FEAT_SME
11	0	INCD, INCH, INCW (vector) — Doubleword	FEAT_SVE || FEAT_SME
11	1	DECD, DECH, DECW (vector) — Doubleword	FEAT_SVE || FEAT_SME

SVE inc/dec register by element count

The encodings in this section are decoded from SVE Element Count.

31							24	23	22	21	20	19			16	15	14	13			11	10	9				5	4				0
0	0	0	0	0	1	0	0	size		1	1	imm4				1	1	1	0	0		D	pattern					Rdn				

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	D	Instruction Details	Feature
00	0	INCB, INCD, INCH, INCW (scalar) — Byte	FEAT_SVE || FEAT_SME
00	1	DECB, DECD, DECH, DECW (scalar) — Byte	FEAT_SVE || FEAT_SME
01	0	INCB, INCD, INCH, INCW (scalar) — Halfword	FEAT_SVE || FEAT_SME
01	1	DECB, DECD, DECH, DECW (scalar) — Halfword	FEAT_SVE || FEAT_SME
10	0	INCB, INCD, INCH, INCW (scalar) — Word	FEAT_SVE || FEAT_SME
10	1	DECB, DECD, DECH, DECW (scalar) — Word	FEAT_SVE || FEAT_SME
11	0	INCB, INCD, INCH, INCW (scalar) — Doubleword	FEAT_SVE || FEAT_SME
11	1	DECB, DECD, DECH, DECW (scalar) — Doubleword	FEAT_SVE || FEAT_SME

SVE saturating inc/dec register by element count

The encodings in this section are decoded from SVE Element Count.

31							24	23 22	21	20	19 16	15			12	11	10	9 5	4 0
0	0	0	0	0	1	0	0	size	1	sf	imm4	1	1	1	1	D	U	pattern	Rdn

size	sf	D	U	Instruction Details	Feature
00	0	0	0	SQINCB — 32-bit	FEAT_SVE || FEAT_SME
00	0	0	1	UQINCB — 32-bit	FEAT_SVE || FEAT_SME
00	0	1	0	SQDECB — 32-bit	FEAT_SVE || FEAT_SME
00	0	1	1	UQDECB — 32-bit	FEAT_SVE || FEAT_SME
00	1	0	0	SQINCB — 64-bit	FEAT_SVE || FEAT_SME
00	1	0	1	UQINCB — 64-bit	FEAT_SVE || FEAT_SME
00	1	1	0	SQDECB — 64-bit	FEAT_SVE || FEAT_SME
00	1	1	1	UQDECB — 64-bit	FEAT_SVE || FEAT_SME
01	0	0	0	SQINCH (scalar) — 32-bit	FEAT_SVE || FEAT_SME
01	0	0	1	UQINCH (scalar) — 32-bit	FEAT_SVE || FEAT_SME
01	0	1	0	SQDECH (scalar) — 32-bit	FEAT_SVE || FEAT_SME
01	0	1	1	UQDECH (scalar) — 32-bit	FEAT_SVE || FEAT_SME
01	1	0	0	SQINCH (scalar) — 64-bit	FEAT_SVE || FEAT_SME
01	1	0	1	UQINCH (scalar) — 64-bit	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	sf	D	U	Instruction Details	Feature
01	1	1	0	SQDECH (scalar) — 64-bit	FEAT_SVE || FEAT_SME
01	1	1	1	UQDECH (scalar) — 64-bit	FEAT_SVE || FEAT_SME
10	0	0	0	SQINCW (scalar) — 32-bit	FEAT_SVE || FEAT_SME
10	0	0	1	UQINCW (scalar) — 32-bit	FEAT_SVE || FEAT_SME
10	0	1	0	SQDECW (scalar) — 32-bit	FEAT_SVE || FEAT_SME
10	0	1	1	UQDECW (scalar) — 32-bit	FEAT_SVE || FEAT_SME
10	1	0	0	SQINCW (scalar) — 64-bit	FEAT_SVE || FEAT_SME
10	1	0	1	UQINCW (scalar) — 64-bit	FEAT_SVE || FEAT_SME
10	1	1	0	SQDECW (scalar) — 64-bit	FEAT_SVE || FEAT_SME
10	1	1	1	UQDECW (scalar) — 64-bit	FEAT_SVE || FEAT_SME
11	0	0	0	SQINCD (scalar) — 32-bit	FEAT_SVE || FEAT_SME
11	0	0	1	UQINCD (scalar) — 32-bit	FEAT_SVE || FEAT_SME
11	0	1	0	SQDECD (scalar) — 32-bit	FEAT_SVE || FEAT_SME
11	0	1	1	UQDECD (scalar) — 32-bit	FEAT_SVE || FEAT_SME
11	1	0	0	SQINCD (scalar) — 64-bit	FEAT_SVE || FEAT_SME
11	1	0	1	UQINCD (scalar) — 64-bit	FEAT_SVE || FEAT_SME
11	1	1	0	SQDECD (scalar) — 64-bit	FEAT_SVE || FEAT_SME
11	1	1	1	UQDECD (scalar) — 64-bit	FEAT_SVE || FEAT_SME

SVE Permute Vector - Extract

SVE Permute Vector - Extract

The encodings in this section are decoded from SVE encodings.

31								23	22	21	20				16	15			13	12															0
0	0	0	0	0	1	0	1	0	op0	1						0	0	0																	

op0	Instruction details	Feature
0	EXT — Destructive	FEAT_SVE || FEAT_SME
1	EXT — Constructive	FEAT_SVE2 || FEAT_SME

SVE Permute Vector - Segments

SVE Permute Vector - Segments

The encodings in this section are decoded from SVE encodings.

31								23	22	21	20				16	15			13	12															0
0	0	0	0	0	1	0	1	1	op0	1						0	0	0																	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0	SVE permute vector segments
1	UNALLOCATED

SVE permute vector segments

The encodings in this section are decoded from SVE Permute Vector - Segments.

31								23	22	21	20			16	15			13	12		11	10	9					5	4						0
0	0	0	0	0	1	0	1	1	0	1	Zm				0	0	0		opc			H	Zn						Zd					

opc	H	Instruction Details	Feature
00	0	ZIP1, ZIP2 (vectors) — Low halves (quadwords)	FEAT_F64MM
00	1	ZIP1, ZIP2 (vectors) — High halves (quadwords)	FEAT_F64MM
01	0	UZP1, UZP2 (vectors) — Even (quadwords)	FEAT_F64MM
01	1	UZP1, UZP2 (vectors) — Odd (quadwords)	FEAT_F64MM
10	x	UNALLOCATED	-
11	0	TRN1, TRN2 (vectors) — Even (quadwords)	FEAT_F64MM
11	1	TRN1, TRN2 (vectors) — Odd (quadwords)	FEAT_F64MM

SVE Bitwise Immediate

SVE Bitwise Immediate

The encodings in this section are decoded from SVE encodings.

31								24	23		22	21		20	19		18	17																	0
0	0	0	0	0	1	0	1	op0				0		0	op1																				

op0	op1	Instruction details	Feature
11	00	DUPM	FEAT_SVE || FEAT_SME
xx	!= 00	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details	Feature
`!=11`	`00`	SVE bitwise logical with immediate (unpredicated)	-

SVE bitwise logical with immediate (unpredicated)

The encodings in this section are decoded from SVE Bitwise Immediate.

31	24	23 22	21 20	19 18	17 5	4 0
0 0 0 0 0 1 0 1		!= 11	0 0	0 0	imm13	Zdn
		opc				

The following constraints also apply to this encoding: opc != '11'

opc	Instruction Details	Feature
`00`	ORR (immediate)	FEAT_SVE || FEAT_SME
`01`	EOR (immediate)	FEAT_SVE || FEAT_SME
`10`	AND (immediate)	FEAT_SVE || FEAT_SME

SVE Integer Wide Immediate - Predicated

SVE Integer Wide Immediate - Predicated

The encodings in this section are decoded from SVE encodings.

31	25	24	23 22	21 20	19 16	15 13	12 0
0 0 0 0 0 1 0		1		0 1		op0	

op0	Instruction details	Feature
`0xx`	SVE copy integer immediate (predicated)	-
`10x`	UNALLOCATED	-
`110`	FCPY	FEAT_SVE || FEAT_SME
`111`	UNALLOCATED	-

SVE copy integer immediate (predicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Predicated.

31	24	23 22	21 20	19 16	15	14	13	12 5	4 0
0 0 0 0 0 1 0	1	size	0 1	Pg	0	M	sh	imm8	Zd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	Instruction Details	Feature
0	CPY (immediate, zeroing)	FEAT_SVE || FEAT_SME
1	CPY (immediate, merging)	FEAT_SVE || FEAT_SME

SVE Permute Vector - One Source Quadwords

SVE Permute Vector - One Source Quadwords

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19		16	15					10	9								0
0	0	0	0	0	1	0	1	op0		1	op1				0	0	1	0	0	1									

op0	op1	Instruction details	Feature
00	x	DUPQ	FEAT_SVE2p1 || FEAT_SME2p1
01	0	EXTQ	FEAT_SVE2p1 || FEAT_SME2p1
01	1	UNALLOCATED	-
1x	x	UNALLOCATED	-

SVE Permute Vector - Unpredicated

SVE Permute Vector - Unpredicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19	18		16	15					10	9	8			5	4	3			0
0	0	0	0	0	1	0	1			1	op0		op1			0	0	1	1	1	0	op2					op3				

op0	op1	op2	op3	Instruction details	Feature
00	000	x	x	DUP (scalar)	FEAT_SVE || FEAT_SME
00	100	x	x	INSR (scalar)	FEAT_SVE || FEAT_SME
00	x01	x	x	UNALLOCATED	-
00	x1x	x	x	UNALLOCATED	-
01	xx0	x	0	SVE move predicate from vector	-
01	xx0	x	1	UNALLOCATED	-
01	xx1	0	x	SVE move predicate into vector	-
01	xx1	1	x	UNALLOCATED	-
10	0xx	x	x	SVE unpack vector elements	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
10	100	x	x	INSR (SIMD&FP scalar)	FEAT_SVE || FEAT_SME
10	101	x	x	UNALLOCATED	-
10	11x	x	x	UNALLOCATED	-
11	000	x	x	REV (vector)	FEAT_SVE || FEAT_SME
11	!= 000	x	x	UNALLOCATED	-

SVE move predicate from vector

The encodings in this section are decoded from SVE Permute Vector - Unpredicated.

31							24	23	22	21	20	19	18	17	16						10	9			5	4	3				0
0	0	0	0	0	1	0	1	opc		1	0	1	opc2		0	0	0	1	1	1	0	Zn				0	Pd				

opc	opc2	Instruction Details	Feature
00	00	UNALLOCATED	-
00	01	PMOV (to predicate) — Byte	FEAT_SVE2p1 || FEAT_SME2p1
00	1x	PMOV (to predicate) — Halfword	FEAT_SVE2p1 || FEAT_SME2p1
01	xx	PMOV (to predicate) — Word	FEAT_SVE2p1 || FEAT_SME2p1
1x	xx	PMOV (to predicate) — Doubleword	FEAT_SVE2p1 || FEAT_SME2p1

SVE move predicate into vector

The encodings in this section are decoded from SVE Permute Vector - Unpredicated.

31							24	23	22	21	20	19	18	17	16						10	9	8		5	4				0
0	0	0	0	0	1	0	1	opc		1	0	1	opc2		1	0	0	1	1	1	0	0	Pn			Zd				

opc	opc2	Instruction Details	Feature
00	00	UNALLOCATED	-
00	01	PMOV (to vector) — Byte	FEAT_SVE2p1 || FEAT_SME2p1
00	1x	PMOV (to vector) — Halfword	FEAT_SVE2p1 || FEAT_SME2p1
01	xx	PMOV (to vector) — Word	FEAT_SVE2p1 || FEAT_SME2p1
1x	xx	PMOV (to vector) — Doubleword	FEAT_SVE2p1 || FEAT_SME2p1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE unpack vector elements

The encodings in this section are decoded from SVE Permute Vector - Unpredicated.

31–24	23–22	21	20–19	18	17	16	15–10	9–5	4–0
0 0 0 0 0 1 0 1	size	1	1 0	0	U	H	0 0 1 1 1 0	Zn	Zd

U	H	Instruction Details	Feature
0	0	SUNPKHI, SUNPKLO — Low half	FEAT_SVE || FEAT_SME
0	1	SUNPKHI, SUNPKLO — High half	FEAT_SVE || FEAT_SME
1	0	UUNPKHI, UUNPKLO — Low half	FEAT_SVE || FEAT_SME
1	1	UUNPKHI, UUNPKLO — High half	FEAT_SVE || FEAT_SME

SVE Permute Predicate

SVE Permute Predicate

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21	20–16	15–13	12–9	8–5	4	3–0
0 0 0 0 0 1 0 1	op0	1	op1	0 1 0	op2		op3	

op0	op1	op2	op3	Instruction details	Feature
00	1000x	0000	0	SVE unpack predicate elements	-
xx	0xxxx	xxx0	0	SVE permute predicate elements	-
xx	10100	0000	0	REV (predicate)	FEAT_SVE || FEAT_SME
xx	10101	0000	0	UNALLOCATED	-
xx	10x0x	0010	0	UNALLOCATED	-
xx	10x0x	01x0	0	UNALLOCATED	-
xx	10x0x	1xx0	0	UNALLOCATED	-
xx	10x1x	xxx0	0	UNALLOCATED	-
xx	11xxx	xxx0	0	UNALLOCATED	-
xx	xxxxx	xxx0	1	UNALLOCATED	-
xx	xxxxx	xxx1	x	UNALLOCATED	-
!=00	1000x	0000	0	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE unpack predicate elements

The encodings in this section are decoded from SVE Permute Predicate.

31–24	23–22	21–17	16	15–13	12–9	8–5	4	3–0
00001010	00	11000	H	010	0000	Pn	0	Pd

H	Instruction Details	Feature
0	PUNPKHI, PUNPKLO — Low half	FEAT_SVE || FEAT_SME
1	PUNPKHI, PUNPKLO — High half	FEAT_SVE || FEAT_SME

SVE permute predicate elements

The encodings in this section are decoded from SVE Permute Predicate.

31–24	23–22	21–20	19–16	15–13	12–11	10	9	8–5	4	3–0
00001010	size	10	Pm	010	opc	H	0	Pn	0	Pd

opc	H	Instruction Details	Feature
00	0	ZIP1, ZIP2 (predicates) — Low halves	FEAT_SVE || FEAT_SME
00	1	ZIP1, ZIP2 (predicates) — High halves	FEAT_SVE || FEAT_SME
01	0	UZP1, UZP2 (predicates) — Even	FEAT_SVE || FEAT_SME
01	1	UZP1, UZP2 (predicates) — Odd	FEAT_SVE || FEAT_SME
10	0	TRN1, TRN2 (predicates) — Even	FEAT_SVE || FEAT_SME
10	1	TRN1, TRN2 (predicates) — Odd	FEAT_SVE || FEAT_SME
11	x	UNALLOCATED	-

SVE Permute Vector - Predicated

SVE Permute Vector - Predicated

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21	20	19–17	16	15–14	13	12–0
00001010		1	op0	op1	op2	10	op3	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
0	000	0	0	CPY (SIMD&FP scalar)	FEAT_SVE || FEAT_SME
0	000	1	0	SVE compress Active elements	-
0	000	x	1	SVE extract element to general register	-
0	001	1	1	UNALLOCATED	-
0	001	x	0	SVE extract element to SIMD&FP scalar register	-
0	01x	x	x	SVE reverse within elements	-
0	100	0	1	CPY (scalar)	FEAT_SVE || FEAT_SME
0	100	x	0	SVE conditionally broadcast element to vector	-
0	101	x	0	SVE conditionally extract element to SIMD&FP scalar	-
0	110	0	0	SPLICE — Destructive	FEAT_SVE || FEAT_SME
0	110	0	1	UNALLOCATED	-
0	110	1	0	SPLICE — Constructive	FEAT_SVE2 || FEAT_SME
0	111	0	x	SVE reverse doublewords	-
0	111	1	0	UNALLOCATED	-
0	1xx	1	1	UNALLOCATED	-
0	x01	0	1	UNALLOCATED	-
1	000	0	0	UNALLOCATED	-
1	000	1	0	EXPAND	FEAT_SVE2p2 || FEAT_SME2p2
1	000	x	1	SVE conditionally extract element to general register	-
1	!= 000	x	x	UNALLOCATED	-

SVE compress Active elements

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23 22	21	20	19 17	16	15	14	13	12 10	9 5	4 0
0	0	0	0	0	1	0	1	size	1	0	0 0 0	1	1	0	0	Pg	Zn	Zd

size	Instruction Details	Feature
0x	COMPACT — Byte and halfword	FEAT_SVE2p2 || FEAT_SME2p2
1x	COMPACT — Word and doubleword	FEAT_SVE || FEAT_SME2p2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE extract element to general register

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19			17	16	15	14	13	12		10	9			5	4			0
0	0	0	0	0	1	0	1	size		1	0	0	0	0		B	1	0	1	Pg			Zn				Rd			

B	Instruction Details	Feature
0	LASTA (scalar)	FEAT_SVE || FEAT_SME
1	LASTB (scalar)	FEAT_SVE || FEAT_SME

SVE extract element to SIMD\&FP scalar register

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19			17	16	15	14	13	12		10	9			5	4			0
0	0	0	0	0	1	0	1	size		1	0	0	0	0	1	B	1	0	0	Pg			Zn				Vd			

B	Instruction Details	Feature
0	LASTA (SIMD&FP scalar)	FEAT_SVE || FEAT_SME
1	LASTB (SIMD&FP scalar)	FEAT_SVE || FEAT_SME

SVE reverse within elements

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19		18	17		16	15	14	13	12		10	9			5	4			0
0	0	0	0	0	1	0	1	size		1	0	0		1	opc			1	0	Z	Pg			Zn				Zd			

opc	Z	Instruction Details	Feature
00	0	REVB, REVH, REVW — Byte, merging	FEAT_SVE || FEAT_SME
00	1	REVB, REVH, REVW — Byte, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	0	REVB, REVH, REVW — Halfword, merging	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Z	Instruction Details	Feature
01	1	REVB, REVH, REVW — Halfword, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	0	REVB, REVH, REVW — Word, merging	FEAT_SVE || FEAT_SME
10	1	REVB, REVH, REVW — Word, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	RBIT — Merging	FEAT_SVE || FEAT_SME
11	1	RBIT — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2

SVE conditionally broadcast element to vector

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23 22	21	20	19		17	16	15	14	13	12 10	9 5	4 0
0	0	0	0	0	1	0	1	size	1	0	1	0	0	B	1	0	0	Pg	Zm	Zdn

B	Instruction Details	Feature
0	CLASTA (vectors)	FEAT_SVE || FEAT_SME
1	CLASTB (vectors)	FEAT_SVE || FEAT_SME

SVE conditionally extract element to SIMD\&FP scalar

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23 22	21	20	19		17	16	15	14	13	12 10	9 5	4 0
0	0	0	0	0	1	0	1	size	1	0	1	0	1	B	1	0	0	Pg	Zm	Vdn

B	Instruction Details	Feature
0	CLASTA (SIMD&FP scalar)	FEAT_SVE || FEAT_SME
1	CLASTB (SIMD&FP scalar)	FEAT_SVE || FEAT_SME

SVE reverse doublewords

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23 22	21	20	19		17	16	15	14	13	12 10	9 5	4 0
0	0	0	0	0	1	0	1	size	1	0	1	1	1	0	1	0	Z	Pg	Zn	Zd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Z	Instruction Details	Feature
00	0	REVD — Merging	FEAT_SME || FEAT_SVE2p1
00	1	REVD — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
!= 00	x	UNALLOCATED	-

SVE conditionally extract element to general register

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19			17	16	15	14	13	12		10	9			5	4			0
0	0	0	0	0	1	0	1	size		1	1	0	0	0		B	1	0	1	Pg			Zm				Rdn			

B	Instruction Details	Feature
0	CLASTA (scalar)	FEAT_SVE || FEAT_SME
1	CLASTB (scalar)	FEAT_SVE || FEAT_SME

SVE Integer Compare - Vectors

SVE Integer Compare - Vectors

The encodings in this section are decoded from SVE encodings.

31							25	24	23	22	21	20			15	14	13													0
0	0	1	0	0	1	0		0			0					op0														

op0	Instruction details
0	SVE integer compare vectors
1	SVE integer compare with wide elements

SVE integer compare vectors

The encodings in this section are decoded from SVE Integer Compare - Vectors.

31							24	23	22	21	20			16	15	14	13	12		10	9			5	4	3			0
0	0	1	0	0	1	0	0	size		0		Zm			op	0	o2	Pg			Zn				ne	Pd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	o2	ne	Instruction Details	Feature
0	0	0	CMP<cc> (vectors) — Higher or same	FEAT_SVE || FEAT_SME
0	0	1	CMP<cc> (vectors) — Higher	FEAT_SVE || FEAT_SME
0	1	0	CMP<cc> (wide elements) — Equal	FEAT_SVE || FEAT_SME
0	1	1	CMP<cc> (wide elements) — Not equal	FEAT_SVE || FEAT_SME
1	0	0	CMP<cc> (vectors) — Greater than or equal	FEAT_SVE || FEAT_SME
1	0	1	CMP<cc> (vectors) — Greater than	FEAT_SVE || FEAT_SME
1	1	0	CMP<cc> (vectors) — Equal	FEAT_SVE || FEAT_SME
1	1	1	CMP<cc> (vectors) — Not equal	FEAT_SVE || FEAT_SME

SVE integer compare with wide elements

The encodings in this section are decoded from SVE Integer Compare - Vectors.

31	24	23 22	21	20 16	15	14	13	12 10	9 5	4	3 0
0 0 1 0 0 1 0 0		size	0	Zm	U	1	lt	Pg	Zn	ne	Pd

U	lt	ne	Instruction Details	Feature
0	0	0	CMP<cc> (wide elements) — Greater than or equal	FEAT_SVE || FEAT_SME
0	0	1	CMP<cc> (wide elements) — Greater than	FEAT_SVE || FEAT_SME
0	1	0	CMP<cc> (wide elements) — Less than	FEAT_SVE || FEAT_SME
0	1	1	CMP<cc> (wide elements) — Less than or equal	FEAT_SVE || FEAT_SME
1	0	0	CMP<cc> (wide elements) — Higher or same	FEAT_SVE || FEAT_SME
1	0	1	CMP<cc> (wide elements) — Higher	FEAT_SVE || FEAT_SME
1	1	0	CMP<cc> (wide elements) — Lower	FEAT_SVE || FEAT_SME
1	1	1	CMP<cc> (wide elements) — Lower or same	FEAT_SVE || FEAT_SME

SVE Propagate Break

SVE Propagate Break

The encodings in this section are decoded from SVE encodings.

31	25	24	23 22	21	20	19 16	15	14	13 10	9	8 0
0 0 1 0 0 1	0	1		0	0		1	1		op0	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0	SVE propagate break from previous partition
1	UNALLOCATED

SVE propagate break from previous partition

The encodings in this section are decoded from SVE Propagate Break.

31							24	23	22	21	20	19			16	15	14	13			10	9	8			5	4	3			0
0	0	1	0	0	1	0	1	op	S	0	0	Pm				1	1	Pg				0	Pn				B	Pd			

op	S	B	Instruction Details	Feature
0	0	0	BRKPA	FEAT_SVE || FEAT_SME
0	0	1	BRKPB	FEAT_SVE || FEAT_SME
0	1	0	BRKPAS	FEAT_SVE || FEAT_SME
0	1	1	BRKPBS	FEAT_SVE || FEAT_SME
1	x	x	UNALLOCATED	-

SVE Partition Break

SVE Partition Break

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19			16	15	14	13			10	9	8			5	4	3			0
0	0	1	0	0	1	0	1	op0		0	1	op1				0	1					op2					op3				

op0	op1	op2	op3	Instruction details
0	1000	0	0	SVE propagate break to next partition
0	1000	0	1	UNALLOCATED
1	1000	0	x	UNALLOCATED
x	0000	0	x	SVE partition break condition
x	x000	1	x	UNALLOCATED
x	x001	x	x	UNALLOCATED
x	x01x	x	x	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details
x	x1xx	x	x	UNALLOCATED

SVE propagate break to next partition

The encodings in this section are decoded from SVE Partition Break.

31							24	23	22	21	20	19			16	15	14	13			10	9	8			5	4	3			0
0	0	1	0	0	1	0	1	0	S	0	1	1	0	0	0	0	1	Pg				0	Pn				0	Pdm			

S	Instruction Details	Feature
0	BRKN	FEAT_SVE || FEAT_SME
1	BRKNS	FEAT_SVE || FEAT_SME

SVE partition break condition

The encodings in this section are decoded from SVE Partition Break.

31							24	23	22	21	20	19			16	15	14	13			10	9	8			5	4	3			0
0	0	1	0	0	1	0	1	B	S	0	1	0	0	0	0	0	1	Pg				0	Pn				M	Pd			

B	S	M	Instruction Details	Feature
x	1	1	UNALLOCATED	-
0	0	x	BRKA	FEAT_SVE || FEAT_SME
0	1	0	BRKAS	FEAT_SVE || FEAT_SME
1	0	x	BRKB	FEAT_SVE || FEAT_SME
1	1	0	BRKBS	FEAT_SVE || FEAT_SME

SVE Predicate Misc

SVE Predicate Misc

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19			16	15	14	13			11	10	9	8			5	4	3			0
0	0	1	0	0	1	0	1			0	1	op0				1	1	op1				op2		op3				op4			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	Instruction details	Feature
0000	xxx	x0	xxxx	0	SVE predicate test	-
0000	xxx	x1	xxxx	0	UNALLOCATED	-
0001	xxx	xx	xxxx	0	UNALLOCATED	-
1000	000	00	xxxx	0	SVE predicate first active	-
1000	000	10	xxxx	0	UNALLOCATED	-
1000	100	10	0000	0	SVE predicate zero	-
1000	100	10	!= 0000	0	UNALLOCATED	-
1000	110	00	xxxx	0	SVE predicate read from FFR (predicated)	-
1001	000	00	xxxx	0	UNALLOCATED	-
1001	000	10	xxxx	0	PNEXT	FEAT_SVE || FEAT_SME
1001	100	10	xxxx	0	UNALLOCATED	-
1001	110	00	0000	0	SVE predicate read from FFR (unpredicated)	-
1001	110	00	!= 0000	0	UNALLOCATED	-
100x	010	x0	xxxx	0	UNALLOCATED	-
100x	0x0	x1	xxxx	0	UNALLOCATED	-
100x	100	0x	xxxx	0	SVE predicate initialize	-
100x	100	11	xxxx	0	UNALLOCATED	-
100x	110	!= 00	xxxx	0	UNALLOCATED	-
100x	xx1	xx	xxxx	0	UNALLOCATED	-
x00x	xxx	xx	xxxx	1	UNALLOCATED	-
x01x	xxx	xx	xxxx	x	UNALLOCATED	-
x1xx	xxx	xx	xxxx	x	UNALLOCATED	-

SVE predicate test

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23	22	21	20	19			16	15	14	13	10	9	8	5	4	3	0
0	0	1	0	0	1	0	1	op	S	0	1	0	0	0	0	1	1	Pg		0	Pn		0	opc2	

op	S	opc2	Instruction Details	Feature
0	0	xxxx	UNALLOCATED	-
0	1	0000	PTEST	FEAT_SVE || FEAT_SME
0	1	!= 0000	UNALLOCATED	-
1	x	xxxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE predicate first active

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23	22	21	20	19			16	15	14	13		11	10	9	8			5	4	3		0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	0	0	0	0	0	Pg				0	Pdn		

op	S	Instruction Details	Feature
0	0	UNALLOCATED	-
0	1	PFIRST	FEAT_SVE || FEAT_SME
1	x	UNALLOCATED	-

SVE predicate zero

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23	22	21	20	19			16	15	14	13		11	10	9	8			5	4	3		0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	Pd		

op	S	Instruction Details	Feature
0	0	PFALSE	FEAT_SVE || FEAT_SME
0	1	UNALLOCATED	-
1	x	UNALLOCATED	-

SVE predicate read from FFR (predicated)

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23	22	21	20	19			16	15	14	13		11	10	9	8			5	4	3		0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	1	1	0	0	0	Pg				0	Pd		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	Instruction Details	Feature
0	0	RDFFR (predicated)	FEAT_SVE
0	1	RDFFRS	FEAT_SVE
1	x	UNALLOCATED	-

SVE predicate read from FFR (unpredicated)

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23	22	21	20	19			16	15	14	13		11	10	9	8				5	4	3		0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	Pd		

op	S	Instruction Details	Feature
0	0	RDFFR (unpredicated)	FEAT_SVE
0	1	UNALLOCATED	-
1	x	UNALLOCATED	-

SVE predicate initialize

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23	22	21					17	16	15	14	13		11	10	9				5	4	3		0
0	0	1	0	0	1	0	1	size		0	1	1	0	0		S	1	1	1	0	0	0	pattern					0	Pd		

S	Instruction Details	Feature
0	PTRUE (predicate)	FEAT_SVE || FEAT_SME
1	PTRUES	FEAT_SVE || FEAT_SME

SVE Predicate Count

SVE Predicate Count

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21		19	18		16	15	14	13		11	10	9	8								0
0	0	1	0	0	1	0	1			1	0	0				1	0	op0				op1									

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details
000	1	SVE predicate count (predicate-as-counter)
xxx	0	SVE predicate count
!= 000	1	UNALLOCATED

SVE predicate count (predicate-as-counter)

The encodings in this section are decoded from SVE Predicate Count.

31							24	23	22	21		19	18		16	15	14	13			11	10	9	8		5	4		0
0	0	1	0	0	1	0	1	size		1	0	0	opc			1	0	0	0	0		vl	1	PNn			Rd		

opc	Instruction Details	Feature
000	CNTP (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
!= 000	UNALLOCATED	-

SVE predicate count

The encodings in this section are decoded from SVE Predicate Count.

31							24	23	22	21		19	18		16	15	14	13			10	9	8		5	4		0
0	0	1	0	0	1	0	1	size		1	0	0	opc			1	0	Pg				0	Pn			Rd		

opc	Instruction Details	Feature
000	CNTP (predicate)	FEAT_SVE || FEAT_SME
001	FIRSTP	FEAT_SVE2p2 || FEAT_SME2p2
010	LASTP	FEAT_SVE2p2 || FEAT_SME2p2
011	UNALLOCATED	-
1xx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE Inc/Dec by Predicate Count

SVE Inc/Dec by Predicate Count

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21		19	18	17	16	15			12	11	10	0
0	0	1	0	0	1	0	1			1	0	1	op0			1	0	0	0	op1		

op0	op1	Instruction details
0	0	SVE saturating inc/dec vector by predicate count
0	1	SVE saturating inc/dec register by predicate count
1	0	SVE inc/dec vector by predicate count
1	1	SVE inc/dec register by predicate count

SVE saturating inc/dec vector by predicate count

The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

31							24	23	22	21		19	18	17	16	15			12	11	10	9	8	5	4	0
0	0	1	0	0	1	0	1	size		1	0	1	0	D	U	1	0	0	0	0	opc		Pm		Zdn	

D	U	opc	Instruction Details	Feature
x	x	!= 00	UNALLOCATED	-
0	0	00	SQINCP (vector)	FEAT_SVE || FEAT_SME
0	1	00	UQINCP (vector)	FEAT_SVE || FEAT_SME
1	0	00	SQDECP (vector)	FEAT_SVE || FEAT_SME
1	1	00	UQDECP (vector)	FEAT_SVE || FEAT_SME

SVE saturating inc/dec register by predicate count

The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

31							24	23	22	21		19	18	17	16	15			12	11	10	9	8	5	4	0
0	0	1	0	0	1	0	1	size		1	0	1	0	D	U	1	0	0	0	1	sf	op	Pm		Rdn	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D	U	sf	op	Instruction Details	Feature
x	x	x	1	UNALLOCATED	-
0	0	0	0	SQINCP (scalar) — 32-bit	FEAT_SVE || FEAT_SME
0	0	1	0	SQINCP (scalar) — 64-bit	FEAT_SVE || FEAT_SME
0	1	0	0	UQINCP (scalar) — 32-bit	FEAT_SVE || FEAT_SME
0	1	1	0	UQINCP (scalar) — 64-bit	FEAT_SVE || FEAT_SME
1	0	0	0	SQDECP (scalar) — 32-bit	FEAT_SVE || FEAT_SME
1	0	1	0	SQDECP (scalar) — 64-bit	FEAT_SVE || FEAT_SME
1	1	0	0	UQDECP (scalar) — 32-bit	FEAT_SVE || FEAT_SME
1	1	1	0	UQDECP (scalar) — 64-bit	FEAT_SVE || FEAT_SME

SVE inc/dec vector by predicate count

The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

31							24	23 22	21		19	18	17	16	15			12	11	10 9	8 5	4 0
0	0	1	0	0	1	0	1	size	1	0	1	1	op	D	1	0	0	0	0	opc2	Pm	Zdn

op	D	opc2	Instruction Details	Feature
0	x	!= 00	UNALLOCATED	-
0	0	00	INCP (vector)	FEAT_SVE || FEAT_SME
0	1	00	DECP (vector)	FEAT_SVE || FEAT_SME
1	x	xx	UNALLOCATED	-

SVE inc/dec register by predicate count

The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

31							24	23 22	21		19	18	17	16	15			12	11	10 9	8 5	4 0
0	0	1	0	0	1	0	1	size	1	0	1	1	op	D	1	0	0	0	1	opc2	Pm	Rdn

op	D	opc2	Instruction Details	Feature
0	x	!= 00	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	D	opc2	Instruction Details	Feature
0	0	00	INCP (scalar)	FEAT_SVE || FEAT_SME
0	1	00	DECP (scalar)	FEAT_SVE || FEAT_SME
1	x	xx	UNALLOCATED	-

SVE Write FFR

SVE Write FFR

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21		19	18	17	16	15			12	11		9	8		5	4				0
0	0	1	0	0	1	0	1			1	0	1	op0	op1		1	0	0	1	op2			op3			op4				

op0	op1	op2	op3	op4	Instruction details
0	00	000	xxxx	00000	SVE FFR write from predicate
1	00	000	0000	00000	SVE FFR initialise
1	00	000	!= 0000	00000	UNALLOCATED
x	00	000	xxxx	!= 00000	UNALLOCATED
x	00	!= 000	xxxx	xxxxx	UNALLOCATED
x	!= 00	xxx	xxxx	xxxxx	UNALLOCATED

SVE FFR write from predicate

The encodings in this section are decoded from SVE Write FFR.

31							24	23	22	21		19	18	17	16	15			12	11		9	8		5	4				0
0	0	1	0	0	1	0	1	opc		1	0	1	0	0	0	1	0	0	1	0	0	0	Pn			0	0	0	0	0

opc	Instruction Details	Feature
00	WRFFR	FEAT_SVE
!= 00	UNALLOCATED	-

SVE FFR initialise

The encodings in this section are decoded from SVE Write FFR.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21		19	18	17	16	15			12	11		9	8			5	4			0
0	0	1	0	0	1	0	1	opc		1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

opc	Instruction Details	Feature
00	SETFFR	FEAT_SVE
!=00	UNALLOCATED	-

SVE Integer Compare - Scalars

SVE Integer Compare - Scalars

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20			16	15	14	13	12	11	10	9			4	3			0
0	0	1	0	0	1	0	1			1					0	0	op0		op1						op2			

op0	op1	op2	Instruction details
0x	xx	xxxx	SVE integer compare scalar count and limit
10	00	0000	SVE conditionally terminate scalars
10	00	!=0000	UNALLOCATED
11	00	xxxx	SVE pointer conflict compare
1x	!=00	xxxx	UNALLOCATED

SVE integer compare scalar count and limit

The encodings in this section are decoded from SVE Integer Compare - Scalars.

31							24	23	22	21	20			16	15			13	12	11	10	9			5	4	3			0
0	0	1	0	0	1	0	1	size		1			Rm		0	0	0		sf	U	lt			Rn		eq		Pd		

U	lt	eq	Instruction Details	Feature
0	0	0	WHILEGE (predicate)	FEAT_SVE2 || FEAT_SME
0	0	1	WHILEGT (predicate)	FEAT_SVE2 || FEAT_SME
0	1	0	WHILELT (predicate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	lt	eq	Instruction Details	Feature
0	1	1	WHILELE (predicate)	FEAT_SVE || FEAT_SME
1	0	0	WHILEHS (predicate)	FEAT_SVE2 || FEAT_SME
1	0	1	WHILEHI (predicate)	FEAT_SVE2 || FEAT_SME
1	1	0	WHILELO (predicate)	FEAT_SVE || FEAT_SME
1	1	1	WHILELS (predicate)	FEAT_SVE || FEAT_SME

SVE conditionally terminate scalars

The encodings in this section are decoded from SVE Integer Compare - Scalars.

31							24	23	22	21	20			16	15	14	13	12	11	10	9				5	4	3				0
0	0	1	0	0	1	0	1	op	sz	1	Rm				0	0	1	0	0	0	Rn					ne	0	0	0	0

op	ne	Instruction Details	Feature
0	x	UNALLOCATED	-
1	0	CTERMEQ, CTERMNE — Equal	FEAT_SVE || FEAT_SME
1	1	CTERMEQ, CTERMNE — Not equal	FEAT_SVE || FEAT_SME

SVE pointer conflict compare

The encodings in this section are decoded from SVE Integer Compare - Scalars.

31							24	23	22	21	20			16	15	14	13	12	11	10	9				5	4	3				0
0	0	1	0	0	1	0	1	size		1	Rm				0	0	1	1	0	0	Rn					rw	Pd				

rw	Instruction Details	Feature
0	WHILEWR	FEAT_SVE2 || FEAT_SME
1	WHILERW	FEAT_SVE2 || FEAT_SME

SVE Scalar Integer Compare - Predicate-as-counter

SVE Scalar Integer Compare - Predicate-as-counter

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20			16	15	14	13			11	10				5	4	3	2			0
0	0	1	0	0	1	0	1			1	op0				0	1	op1				op2					1	op3				

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
00000	110	xxxxxx	x	SVE extract mask predicate from predicate-as-counter	-
00000	111	000000	0	PTRUE (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
00000	111	000000	1	UNALLOCATED	-
00000	111	!= 000000	x	UNALLOCATED	-
xxxxx	01x	xxxxxx	x	SVE integer compare scalar count and limit (predicate pair)	-
xxxxx	x0x	xxxxxx	x	SVE integer compare scalar count and limit (predicate-as-counter)	-
!= 00000	11x	xxxxxx	x	UNALLOCATED	-

SVE extract mask predicate from predicate-as-counter

The encodings in this section are decoded from SVE Scalar Integer Compare - Predicate-as-counter.

31							24	23 22	21	20				16	15	14	13		11	10	8	7	5	4	3	0
0	0	1	0	0	1	0	1	size	1	0	0	0	0	0	0	1	1	1	0	opc		PNn		1	Pd	

opc	Instruction Details	Feature
0xx	PEXT (predicate)	FEAT_SME2 || FEAT_SVE2p1
10x	PEXT (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
11x	UNALLOCATED	-

SVE integer compare scalar count and limit (predicate pair)

The encodings in this section are decoded from SVE Scalar Integer Compare - Predicate-as-counter.

31							24	23 22	21	20				16	15			12	11	10	9		5	4	3	1	0
0	0	1	0	0	1	0	1	size	1			Rm			0	1	0	1	U	lt		Rn		1	Pd		eq

U	lt	eq	Instruction Details	Feature
0	0	0	WHILEGE (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
0	0	1	WHILEGT (predicate pair)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	lt	eq	Instruction Details	Feature
0	1	0	WHILELT (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
0	1	1	WHILELE (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
1	0	0	WHILEHS (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
1	0	1	WHILEHI (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
1	1	0	WHILELO (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
1	1	1	WHILELS (predicate pair)	FEAT_SME2 || FEAT_SVE2p1

SVE integer compare scalar count and limit (predicate-as-counter)

The encodings in this section are decoded from SVE Scalar Integer Compare - Predicate-as-counter.

31							24	23 22	21	20 16	15	14	13	12	11	10	9 5	4	3	2 0
0	0	1	0	0	1	0	1	size	1	Rm	0	1	vl	0	U	lt	Rn	1	eq	PNd

U	lt	eq	Instruction Details	Feature
0	0	0	WHILEGE (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
0	0	1	WHILEGT (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
0	1	0	WHILELT (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
0	1	1	WHILELE (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
1	0	0	WHILEHS (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
1	0	1	WHILEHI (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
1	1	0	WHILELO (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
1	1	1	WHILELS (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1

SVE Integer Wide Immediate - Unpredicated

SVE Integer Wide Immediate - Unpredicated

The encodings in this section are decoded from SVE encodings.

31							24	23 22	21	20 19	18 17	16	15	14	13 0
0	0	1	0	0	1	0	1		1	op0		op1	1	1	

op0	op1	Instruction details
00	x	SVE integer add/subtract immediate (unpredicated)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details
01	x	SVE integer min/max immediate (unpredicated)
10	x	SVE integer multiply immediate (unpredicated)
11	0	SVE broadcast integer immediate (unpredicated)
11	1	SVE broadcast floating-point immediate (unpredicated)

SVE integer add/subtract immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	0 0	opc	1 1	sh	imm8	Zdn

opc	Instruction Details	Feature
000	ADD (immediate)	FEAT_SVE || FEAT_SME
001	SUB (immediate)	FEAT_SVE || FEAT_SME
010	UNALLOCATED	-
011	SUBR (immediate)	FEAT_SVE || FEAT_SME
100	SQADD (immediate)	FEAT_SVE || FEAT_SME
101	UQADD (immediate)	FEAT_SVE || FEAT_SME
110	SQSUB (immediate)	FEAT_SVE || FEAT_SME
111	UQSUB (immediate)	FEAT_SVE || FEAT_SME

SVE integer min/max immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	0 1	opc	1 1	o2	imm8	Zdn

opc	o2	Instruction Details	Feature
0xx	1	UNALLOCATED	-
000	0	SMAX (immediate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	Instruction Details	Feature
001	0	UMAX (immediate)	FEAT_SVE || FEAT_SME
010	0	SMIN (immediate)	FEAT_SVE || FEAT_SME
011	0	UMIN (immediate)	FEAT_SVE || FEAT_SME
1xx	x	UNALLOCATED	-

SVE integer multiply immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	1 0	opc	1 1	o2	imm8	Zdn

opc	o2	Instruction Details	Feature
000	0	MUL (immediate)	FEAT_SVE || FEAT_SME
000	1	UNALLOCATED	-
!= 000	x	UNALLOCATED	-

SVE broadcast integer immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	1 1	opc	0	1 1	sh	imm8	Zd

opc	Instruction Details	Feature
00	DUP (immediate)	FEAT_SVE || FEAT_SME
!= 00	UNALLOCATED	-

SVE broadcast floating-point immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	1 1	opc	1	1 1	o2	imm8	Zd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	Instruction Details	Feature
00	0	FDUP	FEAT_SVE || FEAT_SME
00	1	UNALLOCATED	-
!= 00	x	UNALLOCATED	-

SVE Integer Multiply-Add - Unpredicated

SVE Integer Multiply-Add - Unpredicated

The encodings in this section are decoded from SVE encodings.

31	25	24	23	22	21	20	16	15	14	10	9	0
0 1 0 0 0 1 0		0			0			0	op0			

op0	Instruction details	Feature
0000x	SVE integer dot product (unpredicated)	-
0001x	SVE2 saturating multiply-add interleaved long	-
001xx	CDOT (vectors)	FEAT_SVE2 || FEAT_SME
01xxx	SVE2 complex integer multiply-add	-
10xxx	SVE2 integer multiply-add long	-
110xx	SVE2 saturating multiply-add long	-
1110x	SVE2 saturating multiply-add high	-
11110	SVE mixed sign dot product	-
11111	UNALLOCATED	-

SVE integer dot product (unpredicated)

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31	24	23	22	21	20	16	15	11	10	9	5	4	0
0 1 0 0 0 1 0	0	size		0	Zm		0 0 0 0	0	U	Zn		Zda	

size	U	Instruction Details	Feature
0x	x	UNALLOCATED	-
1x	0	SDOT (4-way, vectors)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	U	Instruction Details	Feature
1x	1	UDOT (4-way, vectors)	FEAT_SVE || FEAT_SME

SVE2 saturating multiply-add interleaved long

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31							24	23	22	21	20				16	15				11	10	9			5	4			0
0	1	0	0	0	1	0	0	size		0	Zm					0	0	0	0	1	S	Zn				Zda			

S	Instruction Details	Feature
0	SQDMLALBT	FEAT_SVE2 || FEAT_SME
1	SQDMLSLBT	FEAT_SVE2 || FEAT_SME

SVE2 complex integer multiply-add

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31							24	23	22	21	20				16	15			13	12	11		10	9			5	4			0
0	1	0	0	0	1	0	0	size		0	Zm					0	0	1		op	rot			Zn				Zda			

op	Instruction Details	Feature
0	CMLA (vectors)	FEAT_SVE2 || FEAT_SME
1	SQRDCMLAH (vectors)	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply-add long

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31							24	23	22	21	20				16	15			13	12	11	10	9			5	4			0
0	1	0	0	0	1	0	0	size		0	Zm					0	1	0		S	U	T	Zn				Zda			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	U	T	Instruction Details	Feature
0	0	0	SMLALB (vectors)	FEAT_SVE2 || FEAT_SME
0	0	1	SMLALT (vectors)	FEAT_SVE2 || FEAT_SME
0	1	0	UMLALB (vectors)	FEAT_SVE2 || FEAT_SME
0	1	1	UMLALT (vectors)	FEAT_SVE2 || FEAT_SME
1	0	0	SMLSLB (vectors)	FEAT_SVE2 || FEAT_SME
1	0	1	SMLSLT (vectors)	FEAT_SVE2 || FEAT_SME
1	1	0	UMLSLB (vectors)	FEAT_SVE2 || FEAT_SME
1	1	1	UMLSLT (vectors)	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply-add long

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31–24	23–22	21	20–16	15–12	11	10	9–5	4–0
0 1 0 0 0 1 0 0	size	0	Zm	0 1 1 0	S	T	Zn	Zda

S	T	Instruction Details	Feature
0	0	SQDMLALB (vectors)	FEAT_SVE2 || FEAT_SME
0	1	SQDMLALT (vectors)	FEAT_SVE2 || FEAT_SME
1	0	SQDMLSLB (vectors)	FEAT_SVE2 || FEAT_SME
1	1	SQDMLSLT (vectors)	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply-add high

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31–24	23–22	21	20–16	15–11	10	9–5	4–0
0 1 0 0 0 1 0 0	size	0	Zm	0 1 1 1 0	S	Zn	Zda

S	Instruction Details	Feature
0	SQRDMLAH (vectors)	FEAT_SVE2 || FEAT_SME
1	SQRDMLSH (vectors)	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE mixed sign dot product

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31–24	23–22	21	20–16	15	14–10	9–5	4–0
01000100	size	0	Zm	0	11110	Zn	Zda

size	Instruction Details	Feature
`10`	USDOT (vectors)	(FEAT_SVE && FEAT_I8MM) || (FEAT_SME && FEAT_I8MM)
`!=10`	UNALLOCATED	-

SVE2 Integer - Predicated

SVE2 Integer - Predicated

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21	20–17	16	15–14	13	12–0
01000100		0	op0		10	op1	

op0	op1	Instruction details
`0010`	`1`	SVE2 integer pairwise add and accumulate long
`0011`	`1`	UNALLOCATED
`011x`	`1`	UNALLOCATED
`0x0x`	`1`	SVE2 integer unary operations (predicated)
`0xxx`	`0`	SVE2 saturating/rounding bitwise shift left (predicated)
`10xx`	`0`	SVE2 integer halving add/subtract (predicated)
`10xx`	`1`	SVE2 integer pairwise arithmetic
`11xx`	`0`	SVE2 saturating add/subtract
`11xx`	`1`	UNALLOCATED

SVE2 integer pairwise add and accumulate long

The encodings in this section are decoded from SVE2 Integer - Predicated.

31–24	23–22	21	20–17	16	15–14	13	12–10	9–5	4–0
01000100	size	0	0010	U	10	1	Pg	Zn	Zda

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Feature
0	SADALP	FEAT_SVE2 || FEAT_SME
1	UADALP	FEAT_SVE2 || FEAT_SME

SVE2 integer unary operations (predicated)

The encodings in this section are decoded from SVE2 Integer - Predicated.

31							24	23	22	21	20	19	18	17	16	15	14	13	12	10	9	5	4	0
0	1	0	0	0	1	0	0	size		0	0	Q	0	Z	op	1	0	1	Pg		Zn		Zd	

Q	Z	op	Instruction Details	Feature
0	0	0	URECPE — Merging	FEAT_SVE2 || FEAT_SME
0	0	1	URSQRTE — Merging	FEAT_SVE2 || FEAT_SME
0	1	0	URECPE — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	1	1	URSQRTE — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	0	0	SQABS — Merging	FEAT_SVE2 || FEAT_SME
1	0	1	SQNEG — Merging	FEAT_SVE2 || FEAT_SME
1	1	0	SQABS — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	1	1	SQNEG — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2

SVE2 saturating/rounding bitwise shift left (predicated)

The encodings in this section are decoded from SVE2 Integer - Predicated.

31							24	23	22	21	20	19	18	17	16	15	14	13	12	10	9	5	4	0
0	1	0	0	0	1	0	0	size		0	0	Q	R	N	U	1	0	0	Pg		Zm		Zdn	

Q	R	N	U	Instruction Details	Feature
0	x	0	x	UNALLOCATED	-
0	0	1	0	SRSHL	FEAT_SVE2 || FEAT_SME
0	0	1	1	URSHL	FEAT_SVE2 || FEAT_SME
0	1	1	0	SRSHLR	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	R	N	U	Instruction Details	Feature
0	1	1	1	URSHLR	FEAT_SVE2 || FEAT_SME
1	0	0	0	SQSHL (vectors)	FEAT_SVE2 || FEAT_SME
1	0	0	1	UQSHL (vectors)	FEAT_SVE2 || FEAT_SME
1	0	1	0	SQRSHL	FEAT_SVE2 || FEAT_SME
1	0	1	1	UQRSHL	FEAT_SVE2 || FEAT_SME
1	1	0	0	SQSHLR	FEAT_SVE2 || FEAT_SME
1	1	0	1	UQSHLR	FEAT_SVE2 || FEAT_SME
1	1	1	0	SQRSHLR	FEAT_SVE2 || FEAT_SME
1	1	1	1	UQRSHLR	FEAT_SVE2 || FEAT_SME

SVE2 integer halving add/subtract (predicated)

The encodings in this section are decoded from SVE2 Integer - Predicated.

31							24	23	22	21		19	18	17	16	15	14	13	12		10	9		5	4		0
0	1	0	0	0	1	0	0	size		0	1	0	R	S	U	1	0	0	Pg			Zm			Zdn		

R	S	U	Instruction Details	Feature
0	0	0	SHADD	FEAT_SVE2 || FEAT_SME
0	0	1	UHADD	FEAT_SVE2 || FEAT_SME
0	1	0	SHSUB	FEAT_SVE2 || FEAT_SME
0	1	1	UHSUB	FEAT_SVE2 || FEAT_SME
1	0	0	SRHADD	FEAT_SVE2 || FEAT_SME
1	0	1	URHADD	FEAT_SVE2 || FEAT_SME
1	1	0	SHSUBR	FEAT_SVE2 || FEAT_SME
1	1	1	UHSUBR	FEAT_SVE2 || FEAT_SME

SVE2 integer pairwise arithmetic

The encodings in this section are decoded from SVE2 Integer - Predicated.

31							24	23	22	21		19	18	17	16	15	14	13	12		10	9		5	4		0
0	1	0	0	0	1	0	0	size		0	1	0	opc		U	1	0	1	Pg			Zm			Zdn		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	U	Instruction Details	Feature
00	0	UNALLOCATED	-
00	1	ADDP	FEAT_SVE2 || FEAT_SME
01	x	UNALLOCATED	-
10	0	SMAXP	FEAT_SVE2 || FEAT_SME
10	1	UMAXP	FEAT_SVE2 || FEAT_SME
11	0	SMINP	FEAT_SVE2 || FEAT_SME
11	1	UMINP	FEAT_SVE2 || FEAT_SME

SVE2 saturating add/subtract

The encodings in this section are decoded from SVE2 Integer - Predicated.

31							24	23 22	21		19	18	17	16	15	14	13	12 10	9 5	4 0
0	1	0	0	0	1	0	0	size	0	1	1	op	S	U	1	0	0	Pg	Zm	Zdn

op	S	U	Instruction Details	Feature
0	0	0	SQADD (vectors, predicated)	FEAT_SVE2 || FEAT_SME
0	0	1	UQADD (vectors, predicated)	FEAT_SVE2 || FEAT_SME
0	1	0	SQSUB (vectors, predicated)	FEAT_SVE2 || FEAT_SME
0	1	1	UQSUB (vectors, predicated)	FEAT_SVE2 || FEAT_SME
1	0	0	SUQADD	FEAT_SVE2 || FEAT_SME
1	0	1	USQADD	FEAT_SVE2 || FEAT_SME
1	1	0	SQSUBR	FEAT_SVE2 || FEAT_SME
1	1	1	UQSUBR	FEAT_SVE2 || FEAT_SME

SVE Multiply - Indexed

SVE Multiply - Indexed

The encodings in this section are decoded from SVE encodings.

31							25	24	23 22	21	20 16	15 10	9 0
0	1	0	0	0	1	0		0		1		op0	

op0	Instruction details
00000x	SVE integer dot product (indexed)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
00001x	SVE2 integer multiply-add (indexed)
00010x	SVE2 saturating multiply-add high (indexed)
00011x	SVE mixed sign dot product (indexed)
001xxx	SVE2 saturating multiply-add (indexed)
0100xx	SVE2 complex integer dot product (indexed)
0101xx	UNALLOCATED
0110xx	SVE2 complex integer multiply-add (indexed)
0111xx	SVE2 complex saturating multiply-add (indexed)
10xxxx	SVE2 integer multiply-add long (indexed)
110xxx	SVE2 integer multiply long (indexed)
1110xx	SVE2 saturating multiply (indexed)
11110x	SVE2 saturating multiply high (indexed)
111110	SVE2 integer multiply (indexed)
111111	UNALLOCATED

SVE integer dot product (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23 22	21	20 16	15				11	10	9 5	4 0
0	1	0	0	0	1	0	0	size	1	opc	0	0	0	0	0	U	Zn	Zda

size	U	Instruction Details	Feature
0x	x	UNALLOCATED	-
10	0	SDOT (4-way, indexed) — 8-bit to 32-bit	FEAT_SVE || FEAT_SME
10	1	UDOT (4-way, indexed) — 8-bit to 32-bit	FEAT_SVE || FEAT_SME
11	0	SDOT (4-way, indexed) — 16-bit to 64-bit	FEAT_SVE || FEAT_SME
11	1	UDOT (4-way, indexed) — 16-bit to 64-bit	FEAT_SVE || FEAT_SME

SVE2 integer multiply-add (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20				16	15					11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		1	opc					0	0	0	0	1		S	Zn					Zda				

size	S	Instruction Details	Feature
0x	0	MLA (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
0x	1	MLS (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
10	0	MLA (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	MLS (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	MLA (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	MLS (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply-add high (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20				16	15					11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		1	opc					0	0	0	1	0		S	Zn					Zda				

size	S	Instruction Details	Feature
0x	0	SQRDMLAH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
0x	1	SQRDMLSH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
10	0	SQRDMLAH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	SQRDMLSH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	SQRDMLAH (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	SQRDMLSH (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE mixed sign dot product (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20				16	15					11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		1	opc					0	0	0	1	1		U	Zn					Zda				

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	U	Instruction Details	Feature
10	0	USDOT (indexed)	(FEAT_SVE && FEAT_I8MM) || (FEAT_SME && FEAT_I8MM)
10	1	SUDOT	(FEAT_SVE && FEAT_I8MM) || (FEAT_SME && FEAT_I8MM)
!=10	x	UNALLOCATED	-

SVE2 saturating multiply-add (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31	24	23 22	21	20 16	15 13	12	11	10	9 5	4 0
0 1 0 0 0 1 0 0		size	1	opc	0 0 1	S	il	T	Zn	Zda

size	S	T	Instruction Details	Feature
0x	x	x	UNALLOCATED	-
10	0	0	SQDMLALB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	0	1	SQDMLALT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	0	SQDMLSLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	1	SQDMLSLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	0	SQDMLALB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	1	SQDMLALT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	0	SQDMLSLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	1	SQDMLSLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 complex integer dot product (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 1 0 0 0 1 0 0		size	1	opc	0 1 0 0	rot	Zn	Zda

size	Instruction Details	Feature
0x	UNALLOCATED	-
10	CDOT (indexed) — 8-bit to 32-bit	FEAT_SVE2 || FEAT_SME
11	CDOT (indexed) — 16-bit to 64-bit	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 complex integer multiply-add (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31–24	23–22	21	20–16	15–12	11–10	9–5	4–0
01000100	size	1	opc	0110	rot	Zn	Zda

size	Instruction Details	Feature
0x	UNALLOCATED	-
10	CMLA (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
11	CMLA (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME

SVE2 complex saturating multiply-add (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31–24	23–22	21	20–16	15–12	11–10	9–5	4–0
01000100	size	1	opc	0111	rot	Zn	Zda

size	Instruction Details	Feature
0x	UNALLOCATED	-
10	SQRDCMLAH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
11	SQRDCMLAH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply-add long (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31–24	23–22	21	20–16	15–14	13	12	11	10	9–5	4–0
01000100	size	1	opc	10	S	U	i1	T	Zn	Zda

size	S	U	T	Instruction Details	Feature
0x	x	x	x	UNALLOCATED	-
10	0	0	0	SMLALB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	S	U	T	Instruction Details	Feature
10	0	0	1	SMLALT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	0	1	0	UMLALB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	0	1	1	UMLALT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	0	0	SMLSLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	0	1	SMLSLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	1	0	UMLSLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	1	1	UMLSLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	0	0	SMLALB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	0	1	SMLALT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	1	0	UMLALB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	1	1	UMLALT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	0	0	SMLSLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	0	1	SMLSLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	1	0	UMLSLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	1	1	UMLSLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply long (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20			16	15		13	12	11	10	9			5	4			0
0	1	0	0	0	1	0	0	size		1	opc				1	1	0	U	il	T	Zn				Zd			

size	U	T	Instruction Details	Feature
0x	x	x	UNALLOCATED	-
10	0	0	SMULLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	0	1	SMULLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	0	UMULLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	1	UMULLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	0	SMULLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	1	SMULLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	0	UMULLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	1	UMULLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20				16	15			12	11	10	9			5	4			0
0	1	0	0	0	1	0	0	size		1	opc					1	1	1	0	i1	T	Zn				Zd			

size	T	Instruction Details	Feature
0x	x	UNALLOCATED	-
10	0	SQDMULLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	SQDMULLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	SQDMULLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	SQDMULLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply high (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20				16	15				11	10	9			5	4			0
0	1	0	0	0	1	0	0	size		1	opc					1	1	1	1	0	R	Zn				Zd			

size	R	Instruction Details	Feature
0x	0	SQDMULH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
0x	1	SQRDMULH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
10	0	SQDMULH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	SQRDMULH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	SQDMULH (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	SQRDMULH (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20				16	15					10	9			5	4			0
0	1	0	0	0	1	0	0	size		1	opc					1	1	1	1	1	0	Zn				Zd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Instruction Details	Feature
0x	MUL (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
10	MUL (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	MUL (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 Widening Integer Arithmetic

SVE2 Widening Integer Arithmetic

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20		16	15	14	13	12		0
0	1	0	0	0	1	0	1			0				0	op0				

op0	Instruction details
0x	SVE2 integer add/subtract long
10	SVE2 integer add/subtract wide
11	SVE2 integer multiply long

SVE2 integer add/subtract long

The encodings in this section are decoded from SVE2 Widening Integer Arithmetic.

31							24	23	22	21	20		16	15	14	13	12	11	10	9		5	4		0
0	1	0	0	0	1	0	1	size		0	Zm			0	0	op	S	U	T	Zn			opc		

op	S	U	T	Instruction Details	Feature
0	0	0	0	SADDLB	FEAT_SVE2 || FEAT_SME
0	0	0	1	SADDLT	FEAT_SVE2 || FEAT_SME
0	0	1	0	UADDLB	FEAT_SVE2 || FEAT_SME
0	0	1	1	UADDLT	FEAT_SVE2 || FEAT_SME
0	1	0	0	SSUBLB	FEAT_SVE2 || FEAT_SME
0	1	0	1	SSUBLT	FEAT_SVE2 || FEAT_SME
0	1	1	0	USUBLB	FEAT_SVE2 || FEAT_SME
0	1	1	1	USUBLT	FEAT_SVE2 || FEAT_SME
1	0	x	x	UNALLOCATED	-
1	1	0	0	SABDLB	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	U	T	Instruction Details	Feature
1	1	0	1	SABDLT	FEAT_SVE2 || FEAT_SME
1	1	1	0	UABDLB	FEAT_SVE2 || FEAT_SME
1	1	1	1	UABDLT	FEAT_SVE2 || FEAT_SME

SVE2 integer add/subtract wide

The encodings in this section are decoded from SVE2 Widening Integer Arithmetic.

31							24	23 22	21	20 16	15	14 13	12	11	10	9 5	4 0
0	1	0	0	0	1	0	1	size	0	Zm	0	1 0	S	U	T	Zn	Zd

S	U	T	Instruction Details	Feature
0	0	0	SADDWB	FEAT_SVE2 || FEAT_SME
0	0	1	SADDWT	FEAT_SVE2 || FEAT_SME
0	1	0	UADDWB	FEAT_SVE2 || FEAT_SME
0	1	1	UADDWT	FEAT_SVE2 || FEAT_SME
1	0	0	SSUBWB	FEAT_SVE2 || FEAT_SME
1	0	1	SSUBWT	FEAT_SVE2 || FEAT_SME
1	1	0	USUBWB	FEAT_SVE2 || FEAT_SME
1	1	1	USUBWT	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply long

The encodings in this section are decoded from SVE2 Widening Integer Arithmetic.

31							24	23 22	21	20 16	15	14 13	12	11	10	9 5	4 0
0	1	0	0	0	1	0	1	size	0	Zm	0	1 1	op	U	T	Zn	Zd

size	op	U	T	Instruction Details	Feature
xx	0	0	0	SQDMULLB (vectors)	FEAT_SVE2 || FEAT_SME
xx	0	0	1	SQDMULLT (vectors)	FEAT_SVE2 || FEAT_SME
xx	1	0	0	SMULLB (vectors)	FEAT_SVE2 || FEAT_SME
xx	1	0	1	SMULLT (vectors)	FEAT_SVE2 || FEAT_SME
xx	1	1	0	UMULLB (vectors)	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	op	U	T	Instruction Details	Feature
xx	1	1	1	UMULLT (vectors)	FEAT_SVE2 || FEAT_SME
00	0	1	0	PMULLB — 128-bit element	FEAT_SVE_PMULL128
00	0	1	1	PMULLT — 128-bit element	FEAT_SVE_PMULL128
!= 00	0	1	0	PMULLB — 16-bit or 64-bit elements	FEAT_SVE2 || FEAT_SME
!= 00	0	1	1	PMULLT — 16-bit or 64-bit elements	FEAT_SVE2 || FEAT_SME

SVE Misc

SVE Misc

The encodings in this section are decoded from SVE encodings.

31–24	23	22	21	20–16	15	14	13–10	9–0
0 1 0 0 0 1 0 1	op0		0		1	0	op1	

op0	op1	Instruction details
0	10xx	SVE2 bitwise shift left long
1	10xx	UNALLOCATED
x	00xx	SVE2 integer add/subtract interleaved long
x	010x	SVE2 bitwise exclusive-OR interleaved
x	0110	SVE integer matrix multiply accumulate
x	0111	UNALLOCATED
x	11xx	SVE2 bitwise permute

SVE2 bitwise shift left long

The encodings in this section are decoded from SVE Misc.

31–24	23	22	21	20–19	18–16	15–12	11	10	9–5	4–0
0 1 0 0 0 1 0 1	0	tszh	0	tszl	imm3	1 0 1 0	U	T	Zn	Zd

U	T	Instruction Details	Feature
0	0	SSHLLB	FEAT_SVE2 || FEAT_SME
0	1	SSHLLT	FEAT_SVE2 || FEAT_SME
1	0	USHLLB	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	T	Instruction Details	Feature
1	1	USHLLT	FEAT_SVE2 || FEAT_SME

SVE2 integer add/subtract interleaved long

The encodings in this section are decoded from SVE Misc.

31	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0 1 0 0 0 1 0 1		size	0	Zm	1 0 0 0	S	tb	Zn	Zd

S	tb	Instruction Details	Feature
0	0	SADDLBT	FEAT_SVE2 || FEAT_SME
0	1	UNALLOCATED	-
1	0	SSUBLBT	FEAT_SVE2 || FEAT_SME
1	1	SSUBLTB	FEAT_SVE2 || FEAT_SME

SVE2 bitwise exclusive-OR interleaved

The encodings in this section are decoded from SVE Misc.

31	24	23 22	21	20 16	15 11	10	9 5	4 0
0 1 0 0 0 1 0 1		size	0	Zm	1 0 0 1 0	tb	Zn	Zd

tb	Instruction Details	Feature
0	EORBT	FEAT_SVE2 || FEAT_SME
1	EORTB	FEAT_SVE2 || FEAT_SME

SVE integer matrix multiply accumulate

The encodings in this section are decoded from SVE Misc.

31	24	23 22	21	20 16	15 14	13 10	9 5	4 0
0 1 0 0 0 1 0 1		uns	0	Zm	1 0	0 1 1 0	Zn	Zd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

uns	Instruction Details	Feature
00	SMMLA	FEAT_SVE && FEAT_I8MM
01	UNALLOCATED	-
10	USMMLA	FEAT_SVE && FEAT_I8MM
11	UMMLA	FEAT_SVE && FEAT_I8MM

SVE2 bitwise permute

The encodings in this section are decoded from SVE Misc.

31	24	23	22	21	20	16	15	12	11	10	9	5	4	0
0 1 0 0 0 1 0 1		size		0	Zm		1 0 1 1		opc		Zn		Zd	

opc	Instruction Details	Feature
00	BEXT	FEAT_SVE_BitPerm
01	BDEP	FEAT_SVE_BitPerm
10	BGRP	FEAT_SVE_BitPerm
11	UNALLOCATED	-

SVE2 Accumulate

SVE2 Accumulate

The encodings in this section are decoded from SVE encodings.

31	24	23	22	21	20	17	16	15	14	13	11	10	0
0 1 0 0 0 1 0 1				0	op0			1	1	op1			

op0	op1	Instruction details
0000	011	SVE2 complex integer add
xxxx	00x	SVE2 integer absolute difference and accumulate long
xxxx	010	SVE2 integer add/subtract long with carry
xxxx	10x	SVE2 bitwise shift right and accumulate
xxxx	110	SVE2 bitwise shift and insert
xxxx	111	SVE2 integer absolute difference and accumulate
!= 0000	011	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 complex integer add

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20				17	16	15	14	13			11	10	9				5	4				0
0	1	0	0	0	1	0	1	size		0	0	0	0	0		op	1	1	0	1	1		rot	Zm						Zdn			

op	Instruction Details	Feature
0	CADD	FEAT_SVE2 || FEAT_SME
1	SQCADD	FEAT_SVE2 || FEAT_SME

SVE2 integer absolute difference and accumulate long

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20					16	15				12	11	10	9				5	4				0
0	1	0	0	0	1	0	1	size		0	Zm						1	1	0	0		U	T	Zn						Zda			

U	T	Instruction Details	Feature
0	0	SABALB	FEAT_SVE2 || FEAT_SME
0	1	SABALT	FEAT_SVE2 || FEAT_SME
1	0	UABALB	FEAT_SVE2 || FEAT_SME
1	1	UABALT	FEAT_SVE2 || FEAT_SME

SVE2 integer add/subtract long with carry

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20					16	15		14	13			11	10	9				5	4				0
0	1	0	0	0	1	0	1	size		0	Zm						1		1	0	1		0	T	Zn						Zda			

size	T	Instruction Details	Feature
0x	0	ADCLB	FEAT_SVE2 || FEAT_SME
0x	1	ADCLT	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	T	Instruction Details	Feature
1x	0	SBCLB	FEAT_SVE2 || FEAT_SME
1x	1	SBCLT	FEAT_SVE2 || FEAT_SME

SVE2 bitwise shift right and accumulate

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20	19	18		16	15			12	11	10	9		5	4		0
0	1	0	0	0	1	0	1	tszh		0	tszl		imm3			1	1	1	0	R	U	Zn			Zda		

R	U	Instruction Details	Feature
0	0	SSRA	FEAT_SVE2 || FEAT_SME
0	1	USRA	FEAT_SVE2 || FEAT_SME
1	0	SRSRA	FEAT_SVE2 || FEAT_SME
1	1	URSRA	FEAT_SVE2 || FEAT_SME

SVE2 bitwise shift and insert

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20	19	18		16	15	14	13			11	10	9		5	4		0
0	1	0	0	0	1	0	1	tszh		0	tszl		imm3			1	1	1	1		0	op	Zn			Zd	

op	Instruction Details	Feature
0	SRI	FEAT_SVE2 || FEAT_SME
1	SLI	FEAT_SVE2 || FEAT_SME

SVE2 integer absolute difference and accumulate

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20				16	15	14	13			11	10	9		5	4		0
0	1	0	0	0	1	0	1	size		0	Zm					1	1	1	1		1	U	Zn			Zda		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Feature
0	SABA	FEAT_SVE2 || FEAT_SME
1	UABA	FEAT_SVE2 || FEAT_SME

SVE2 Narrowing

SVE2 Narrowing

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	6	5	4	0
0	1	0	0	0	1	0	1	op0		1			op1		op2	0	op3				op4			op5		

op0	op1	op2	op3	op4	op5	Instruction details
0	00	0	10	x	x	SVE2 saturating extract narrow
0	00	1	10	0	0	SME2 multi-vec extract narrow
0	00	1	10	0	1	UNALLOCATED
0	00	1	10	1	x	UNALLOCATED
0	xx	x	0x	x	x	SVE2 bitwise shift right narrow
0	!=00	x	10	x	x	UNALLOCATED
1	xx	x	0x	0	0	SME2 multi-vec shift narrow
1	xx	x	0x	0	1	UNALLOCATED
1	xx	x	0x	1	x	UNALLOCATED
1	xx	x	10	x	x	UNALLOCATED
x	xx	x	11	x	x	SVE2 integer add/subtract narrow high part

SVE2 saturating extract narrow

The encodings in this section are decoded from SVE2 Narrowing.

31							24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	0	0	1	0	1	0		1	tszl		0	0	0	0	1	0	opc		T	Zn		Zd	
									tszh																

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	T	Instruction Details	Feature
00	0	SQXTNB	FEAT_SVE2 || FEAT_SME
00	1	SQXTNT	FEAT_SVE2 || FEAT_SME
01	0	UQXTNB	FEAT_SVE2 || FEAT_SME
01	1	UQXTNT	FEAT_SVE2 || FEAT_SME
10	0	SQXTUNB	FEAT_SVE2 || FEAT_SME
10	1	SQXTUNT	FEAT_SVE2 || FEAT_SME
11	x	UNALLOCATED	-

SME2 multi-vec extract narrow

The encodings in this section are decoded from SVE2 Narrowing.

tszh	tszl	opc	Instruction Details	Feature
0	10	00	SQCVTN	FEAT_SME2 || FEAT_SVE2p1
0	10	01	UQCVTN	FEAT_SME2 || FEAT_SVE2p1
0	10	10	SQCVTUN	FEAT_SME2 || FEAT_SVE2p1
0	10	11	UNALLOCATED	-
0	!=10	xx	UNALLOCATED	-
1	xx	xx	UNALLOCATED	-

SVE2 bitwise shift right narrow

The encodings in this section are decoded from SVE2 Narrowing.

op	U	R	T	Instruction Details	Feature
0	0	0	0	SQSHRUNB	FEAT_SVE2 || FEAT_SME
0	0	0	1	SQSHRUNT	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	R	T	Instruction Details	Feature
0	0	1	0	SQRSHRUNB	FEAT_SVE2 || FEAT_SME
0	0	1	1	SQRSHRUNT	FEAT_SVE2 || FEAT_SME
0	1	0	0	SHRNB	FEAT_SVE2 || FEAT_SME
0	1	0	1	SHRNT	FEAT_SVE2 || FEAT_SME
0	1	1	0	RSHRNB	FEAT_SVE2 || FEAT_SME
0	1	1	1	RSHRNT	FEAT_SVE2 || FEAT_SME
1	0	0	0	SQSHRNB	FEAT_SVE2 || FEAT_SME
1	0	0	1	SQSHRNT	FEAT_SVE2 || FEAT_SME
1	0	1	0	SQRSHRNB	FEAT_SVE2 || FEAT_SME
1	0	1	1	SQRSHRNT	FEAT_SVE2 || FEAT_SME
1	1	0	0	UQSHRNB	FEAT_SVE2 || FEAT_SME
1	1	0	1	UQSHRNT	FEAT_SVE2 || FEAT_SME
1	1	1	0	UQRSHRNB	FEAT_SVE2 || FEAT_SME
1	1	1	1	UQRSHRNT	FEAT_SVE2 || FEAT_SME

SME2 multi-vec shift narrow

The encodings in this section are decoded from SVE2 Narrowing.

31								23	22	21	20				16	15	14	13	12	11	10	9				6	5	4				0
0	1	0	0	0	1	0	1	1	op0	1	opc					0	0	op1	U	R	0	Zn					0	Zd				

op0	opc	op1	U	R	Instruction Details	Feature
0	0xxxx	x	x	x	UNALLOCATED	-
0	1xxxx	x	x	0	UNALLOCATED	-
0	1xxxx	0	0	1	SQRSHRUN	FEAT_SME2 || FEAT_SVE2p1
0	1xxxx	0	1	1	UNALLOCATED	-
0	1xxxx	1	0	1	SQRSHRN	FEAT_SME2 || FEAT_SVE2p1
0	1xxxx	1	1	1	UQRSHRN	FEAT_SME2 || FEAT_SVE2p1
1	xxxxx	x	x	x	UNALLOCATED	-

SVE2 integer add/subtract narrow high part

The encodings in this section are decoded from SVE2 Narrowing.

31								24	23	22	21	20				16	15	14	13	12	11	10	9				5	4				0
0	1	0	0	0	1	0	1	size			1	Zm					0	1	1	S	R	T	Zn					Zd				

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	R	T	Instruction Details	Feature
0	0	0	ADDHNB	FEAT_SVE2 || FEAT_SME
0	0	1	ADDHNT	FEAT_SVE2 || FEAT_SME
0	1	0	RADDHNB	FEAT_SVE2 || FEAT_SME
0	1	1	RADDHNT	FEAT_SVE2 || FEAT_SME
1	0	0	SUBHNB	FEAT_SVE2 || FEAT_SME
1	0	1	SUBHNT	FEAT_SVE2 || FEAT_SME
1	1	0	RSUBHNB	FEAT_SVE2 || FEAT_SME
1	1	1	RSUBHNT	FEAT_SVE2 || FEAT_SME

SVE2 Histogram Computation (Segment) and Lookup Table

SVE2 Histogram Computation (Segment) and Lookup Table

The encodings in this section are decoded from SVE encodings.

31–24	23	22	21	20–16	15–13	12–10	9–0
01000101		op0	1		101	op1	

op0	op1	Instruction details	Feature
0	0x1	UNALLOCATED	-
1	001	LUTI4 (8-bit and 16-bit) — Byte, single register table	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)
1	011	UNALLOCATED	-
x	000	HISTSEG	FEAT_SVE2
x	100	LUTI2 (8-bit and 16-bit) — Byte	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)
x	1x1	SVE2 lookup table with 4-bit indices and 16-bit element size	-
x	x10	LUTI2 (8-bit and 16-bit) — Halfword	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)

SVE2 lookup table with 4-bit indices and 16-bit element size

The encodings in this section are decoded from SVE2 Histogram Computation (Segment) and Lookup Table.

31–24	23–22	21	20–16	15–12	11	10	9–5	4–0
01000101	i2	1	Zm	1011	op	1	Zn	Zd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Feature
0	LUTI4 (8-bit and 16-bit) — Halfword, two register table	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)
1	LUTI4 (8-bit and 16-bit) — Halfword, single register table	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)

SVE2 Crypto Extensions

SVE2 Crypto Extensions

The encodings in this section are decoded from SVE encodings.

31	24	23	22	21	20	18	17	16	15	13	12	10	9	5	4	1	0
0 1 0 0 0 1 0 1				1	op0		op1		1 1 1		op2		op3				op4

op0	op1	op2	op3	op4	Instruction details
000	00	00x	00000	x	SVE2 crypto unary operations
000	00	00x	!= 00000	x	UNALLOCATED
000	01	00x	xxxxx	x	UNALLOCATED
000	1x	00x	xxxxx	x	SVE2 crypto destructive binary operations
xx0	1x	01x	xxxxx	x	SVE2 multi-vector AES single round (two registers)
xx1	1x	01x	xxxxx	x	SVE2 multi-vector AES single round (four registers)
xxx	00	01x	xxxxx	x	UNALLOCATED
xxx	01	01x	xxxxx	x	UNALLOCATED
xxx	xx	10x	xxxxx	x	SVE2 crypto constructive binary operations
xxx	xx	110	xxxxx	0	SVE2 Multi-vector polynomial multiply long
xxx	xx	111	xxxxx	0	SVE2 Multi-vector polynomial multiply long and accumulate vectors
xxx	xx	11x	xxxxx	1	UNALLOCATED
!= 000	xx	00x	xxxxx	x	UNALLOCATED

SVE2 crypto unary operations

The encodings in this section are decoded from SVE2 Crypto Extensions.

31	24	23	22	21	20	18	17	16	15	11	10	9	5	4	0
0 1 0 0 0 1 0 1		size		1	0 0 0		0	0	1 1 1 0 0		op	0 0 0 0 0		Zdn	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	op	Instruction Details	Feature
00	0	AESMC	FEAT_SVE_AES
00	1	AESIMC	FEAT_SVE_AES
!= 00	x	UNALLOCATED	-

SVE2 crypto destructive binary operations

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23 22	21	20			18	17	16	15					11	10	9			5	4			0
0	1	0	0	0	1	0	1	size	1	0	0	0		1	op	1	1	1	0	0		o2	Zm				Zdn			

size	op	o2	Instruction Details	Feature
00	0	0	AESE (vectors)	FEAT_SVE_AES
00	0	1	AESD (vectors)	FEAT_SVE_AES
00	1	0	SM4E	FEAT_SVE_SM4
00	1	1	UNALLOCATED	-
!= 00	x	x	UNALLOCATED	-

SVE2 multi-vector AES single round (two registers)

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23 22	21	20	19	18	17	16	15					11	10	9			5	4		1	0
0	1	0	0	0	1	0	1	size	1	i2		0	1	op	1	1	1	0	1		o2	Zm				Zdn			o3

size	op	o2	o3	Instruction Details	Feature
00	x	x	1	UNALLOCATED	-
00	0	0	0	AESE (indexed)	FEAT_SVE_AES2
00	0	1	0	AESD (indexed)	FEAT_SVE_AES2
00	1	0	0	AESEMC	FEAT_SVE_AES2
00	1	1	0	AESDIMC	FEAT_SVE_AES2
!= 00	x	x	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 multi-vector AES single round (four registers)

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20	19	18	17	16	15					11	10	9			5	4			2	1	0
0	1	0	0	0	1	0	1	size		1	i2		1	1	op	1	1	1	0	1		o2	Zm				Zdn				opc3	

size	op	o2	opc3	Instruction Details	Feature
00	x	x	!= 00	UNALLOCATED	-
00	0	0	00	AESE (indexed)	FEAT_SVE_AES2
00	0	1	00	AESD (indexed)	FEAT_SVE_AES2
00	1	0	00	AESEMC	FEAT_SVE_AES2
00	1	1	00	AESDIMC	FEAT_SVE_AES2
!= 00	x	x	xx	UNALLOCATED	-

SVE2 crypto constructive binary operations

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20				16	15					11	10	9			5	4				0
0	1	0	0	0	1	0	1	size		1	Zm					1	1	1	1	0		op	Zn				Zd				

size	op	Instruction Details	Feature
00	0	SM4EKEY	FEAT_SVE_SM4
00	1	RAX1	FEAT_SVE_SHA3
!= 00	x	UNALLOCATED	-

SVE2 Multi-vector polynomial multiply long

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20				16	15			13	12			10	9			5	4			1	0
0	1	0	0	0	1	0	1	size		1	Zm					1	1	1		1	1	0		Zn				Zd				0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Instruction Details	Feature
00	PMULL	FEAT_SVE_AES2
!= 00	UNALLOCATED	-

SVE2 Multi-vector polynomial multiply long and accumulate vectors

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20				16	15		13	12		10	9			5	4			1	0
0	1	0	0	0	1	0	1	size		1	Zm					1	1	1	1	1	1	Zn				Zda				0

size	Instruction Details	Feature
00	PMLAL	FEAT_SVE_AES2
!= 00	UNALLOCATED	-

SVE2 FP8 widening multiply-add

SVE2 FP8 widening multiply-add

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20				16	15	14	13	12	11	10	9							0
0	1	1	0	0	1	0	0	op0	0	1						1	0	op1		1	0								

op0	op1	Instruction details
0	x	SVE2 FP8 multiply-add long long
1	0	SVE2 FP8 multiply-add long
1	1	UNALLOCATED

SVE2 FP8 multiply-add long long

The encodings in this section are decoded from SVE2 FP8 widening multiply-add.

31							24	23	22	21	20				16	15	14	13		12	11	10	9				5	4			0
0	1	1	0	0	1	0	0	0	0	1	Zm					1	0	TT			1	0	Zn					Zda			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TT	Instruction Details	Feature
00	FMLALLBB (vectors)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
01	FMLALLBT (vectors)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
10	FMLALLTB (vectors)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
11	FMLALLTT (vectors)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)

SVE2 FP8 multiply-add long

The encodings in this section are decoded from SVE2 FP8 widening multiply-add.

31							24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	1	1	0	0	1	0	0	1	0	1	Zm				1	0	0	T	1	0	Zn				Zda			

T	Instruction Details	Feature
0	FMLALB (vectors, FP8 to FP16)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
1	FMLALT (vectors, FP8 to FP16)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)

SVE2 floating-point unary operations - zeroing predicated

SVE2 floating-point unary operations - zeroing predicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21			19	18			16	15	14										0
0	1	1	0	0	1	0	0			0	1	1		op0				1											

op0	Instruction details
00x	Floating-point round to integral value (predicated)
010	Floating-point convert (predicated)
011	Floating-point square root (predicated)
10x	Floating-point round and convert from integer (predicated)
11x	Floating-point log and convert to integer

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Floating-point round to integral value (predicated)

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23	22	21					17	16	15	14	13	12			10	9					5	4					0
0	1	1	0	0	1	0	0	size		0	1	1	0	0		op	1	opc2		Pg				Zn						Zd					

op	opc2	Instruction Details	Feature
0	00	FRINT<r> — Nearest with ties to even, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	01	FRINT<r> — Toward plus infinity, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	10	FRINT<r> — Toward minus infinity, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	11	FRINT<r> — Toward zero, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	00	FRINT<r> — Nearest with ties to away, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	01	UNALLOCATED	-
1	10	FRINT<r> — Current mode signalling inexact, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	11	FRINT<r> — Current mode, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Floating-point convert (predicated)

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23	22	21		19	18		16	15	14	13	12			10	9					5	4					0
0	1	1	0	0	1	0	0	opc		0	1	1	0	1	0	1	opc2			Pg				Zn						Zd					

opc	opc2	Instruction Details	Feature
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTX	FEAT_SVE2p2 || FEAT_SME2p2
01	xx	UNALLOCATED	-
10	00	FCVT — Single-precision to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	01	FCVT — Half-precision to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	10	BFCVT	FEAT_SVE2p2 || FEAT_SME2p2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	Instruction Details	Feature
11	00	FCVT — Double-precision to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	01	FCVT — Half-precision to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	10	FCVT — Double-precision to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	11	FCVT — Single-precision to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Floating-point square root (predicated)

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23 22	21		19	18		16	15	14 13	12 10	9 5	4 0
0	1	1	0	0	1	0	0	size	0	1	1	0	1	1	1	opc	Pg	Zn	Zd

opc	Instruction Details	Feature
00	FRECPX	FEAT_SVE2p2 || FEAT_SME2p2
01	FSQRT	FEAT_SVE2p2 || FEAT_SME2p2
1x	UNALLOCATED	-

Floating-point round and convert from integer (predicated)

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23 22	21				17	16	15	14	13	12 10	9 5	4 0
0	1	1	0	0	1	0	0	opc	0	1	1	1	0	o2	1	o3	U	Pg	Zn	Zd

opc	o2	o3	U	Instruction Details	Feature
00	0	x	0	FRINT32Z	FEAT_SVE2p2 || FEAT_SME2p2
00	0	x	1	FRINT32X	FEAT_SVE2p2 || FEAT_SME2p2
00	1	x	0	FRINT64Z	FEAT_SVE2p2 || FEAT_SME2p2
00	1	x	1	FRINT64X	FEAT_SVE2p2 || FEAT_SME2p2
01	0	0	x	UNALLOCATED	-
01	0	1	0	SCVTF (predicated) — 16-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	o3	U	Instruction Details	Feature
01	0	1	1	UCVTF (predicated) — 16-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	0	0	SCVTF (predicated) — 32-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	0	1	UCVTF (predicated) — 32-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	1	0	SCVTF (predicated) — 64-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	1	1	UCVTF (predicated) — 64-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	0	x	x	UNALLOCATED	-
10	1	0	0	SCVTF (predicated) — 32-bit to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	1	0	1	UCVTF (predicated) — 32-bit to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	1	1	x	UNALLOCATED	-
11	0	0	0	SCVTF (predicated) — 32-bit to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	0	1	UCVTF (predicated) — 32-bit to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	1	x	UNALLOCATED	-
11	1	0	0	SCVTF (predicated) — 64-bit to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	0	1	UCVTF (predicated) — 64-bit to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	1	0	SCVTF (predicated) — 64-bit to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	1	1	UCVTF (predicated) — 64-bit to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Floating-point log and convert to integer

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23 22	21				17	16	15	14	13	12 10	9 5	4 0
0	1	1	0	0	1	0	0	opc	0	1	1	1	1	o2	1	o3	U	Pg	Zn	Zd

opc	o2	o3	U	Instruction Details	Feature
00	0	x	x	FLOGB	FEAT_SVE2p2 || FEAT_SME2p2
00	1	x	x	UNALLOCATED	-
01	0	0	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	o3	U	Instruction Details	Feature
01	0	1	0	FCVTZS — Half-precision to 16-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	0	1	1	FCVTZU — Half-precision to 16-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	0	0	FCVTZS — Half-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	0	1	FCVTZU — Half-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	1	0	FCVTZS — Half-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	1	1	FCVTZU — Half-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	0	x	x	UNALLOCATED	-
10	1	0	0	FCVTZS — Single-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	1	0	1	FCVTZU — Single-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	1	1	x	UNALLOCATED	-
11	0	0	0	FCVTZS — Double-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	0	1	FCVTZU — Double-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	1	x	UNALLOCATED	-
11	1	0	0	FCVTZS — Single-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	0	1	FCVTZU — Single-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	1	0	FCVTZS — Double-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	1	1	FCVTZU — Double-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

SVE floating-point widening multiply-add - indexed

SVE floating-point widening multiply-add - indexed

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20			16	15	14	13	12	11									0
0	1	1	0	0	1	0	0	op0		1					0	1	op1	0										

op0	op1	Instruction details
0	0	SVE BFloat16 floating-point dot product (indexed)
0	1	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details
1	x	SVE floating-point multiply-add long (indexed)

SVE BFloat16 floating-point dot product (indexed)

The encodings in this section are decoded from SVE floating-point widening multiply-add - indexed.

31							24	23	22	21	20 19	18 16	15 14	13	12	11 10	9 5	4 0
0	1	1	0	0	1	0	0	0	op	1	i2	Zm	0 1	0	0	opc2	Zn	Zda

op	opc2	Instruction Details	Feature
0	x1	FDOT (2-way, indexed, FP8 to FP16)	FEAT_SSVE_FP8DOT2 || (FEAT_SVE2 && FEAT_FP8DOT2)
0	00	FDOT (2-way, indexed, FP16 to FP32)	FEAT_SME2 || FEAT_SVE2p1
0	10	UNALLOCATED	-
1	00	BFDOT (indexed)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	01	FDOT (4-way, indexed)	FEAT_SSVE_FP8DOT4 || (FEAT_SVE2 && FEAT_FP8DOT4)
1	1x	UNALLOCATED	-

SVE floating-point multiply-add long (indexed)

The encodings in this section are decoded from SVE floating-point widening multiply-add - indexed.

31							24	23	22	21	20 19	18 16	15 14	13	12	11	10	9 5	4 0
0	1	1	0	0	1	0	0	1	o2	1	i3h	Zm	0 1	op	0	i3l	T	Zn	Zda

o2	op	T	Instruction Details	Feature
0	0	0	FMLALB (indexed, FP16 to FP32)	FEAT_SVE2 || FEAT_SME
0	0	1	FMLALT (indexed, FP16 to FP32)	FEAT_SVE2 || FEAT_SME
0	1	0	FMLSLB (indexed)	FEAT_SVE2 || FEAT_SME
0	1	1	FMLSLT (indexed)	FEAT_SVE2 || FEAT_SME
1	0	0	BFMLALB (indexed)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	0	1	BFMLALT (indexed)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	1	0	BFMLSLB (indexed)	FEAT_SME2 || FEAT_SVE2p1
1	1	1	BFMLSLT (indexed)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE floating-point widening multiply-add

SVE floating-point widening multiply-add

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20			16	15	14	13	12	11	10						0
0	1	1	0	0	1	0	0	op0		1					1	0	op1	0	0							

op0	op1	Instruction details
0	0	SVE BFloat16 floating-point dot product
0	1	UNALLOCATED
1	x	SVE floating-point multiply-add long

SVE BFloat16 floating-point dot product

The encodings in this section are decoded from SVE floating-point widening multiply-add.

31							24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	1	1	0	0	1	0	0	0	op	1	Zm				1	0	0	0	0	o2	Zn				Zda			

op	o2	Instruction Details	Feature
0	0	FDOT (2-way, vectors, FP16 to FP32)	FEAT_SME2 || FEAT_SVE2p1
0	1	FDOT (2-way, vectors, FP8 to FP16)	FEAT_SSVE_FP8DOT2 || (FEAT_SVE2 && FEAT_FP8DOT2)
1	0	BFDOT (vectors)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	1	FDOT (4-way, vectors)	FEAT_SSVE_FP8DOT4 || (FEAT_SVE2 && FEAT_FP8DOT4)

SVE floating-point multiply-add long

The encodings in this section are decoded from SVE floating-point widening multiply-add.

31							24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	1	1	0	0	1	0	0	1	o2	1	Zm				1	0	op	0	0	T	Zn				Zda			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o2	op	T	Instruction Details	Feature
0	0	0	FMLALB (vectors, FP16 to FP32)	FEAT_SVE2 || FEAT_SME
0	0	1	FMLALT (vectors, FP16 to FP32)	FEAT_SVE2 || FEAT_SME
0	1	0	FMLSLB (vectors)	FEAT_SVE2 || FEAT_SME
0	1	1	FMLSLT (vectors)	FEAT_SVE2 || FEAT_SME
1	0	0	BFMLALB (vectors)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	0	1	BFMLALT (vectors)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	1	0	BFMLSLB (vectors)	FEAT_SME2 || FEAT_SVE2p1
1	1	1	BFMLSLT (vectors)	FEAT_SME2 || FEAT_SVE2p1

SVE floating-point unary operations - unpredicated

SVE floating-point unary operations - unpredicated

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21–19	18–16	15–12	11–10	9–6	5	4–0
0 1 1 0 0 1 0 1	op0	0 0 1	op1	0 0 1 1	op2		op3	

op0	op1	op2	op3	Instruction details
00	00x	xx	x	SVE2 FP8 upconverts
00	010	xx	0	SVE2 FP8 downconverts
00	010	xx	1	UNALLOCATED
00	011	xx	x	UNALLOCATED
xx	10x	xx	x	UNALLOCATED
xx	11x	00	x	SVE floating-point reciprocal estimate (unpredicated)
xx	11x	!= 00	x	UNALLOCATED
!= 00	0xx	xx	x	UNALLOCATED

SVE2 FP8 upconverts

The encodings in this section are decoded from SVE floating-point unary operations - unpredicated.

31–24	23–22	21–17	16	15–12	11–10	9–5	4–0
0 1 1 0 0 1 0 1	0 0	0 0 1 0 0	L	0 0 1 1	opc	Zn	Zd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	opc	Instruction Details	Feature
0	00	F1CVT, F2CVT — F1CVT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
0	01	F1CVT, F2CVT — F2CVT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
0	10	BF1CVT, BF2CVT — BF1CVT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
0	11	BF1CVT, BF2CVT — BF2CVT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
1	00	F1CVTLT, F2CVTLT — F1CVTLT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
1	01	F1CVTLT, F2CVTLT — F2CVTLT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
1	10	BF1CVTLT, BF2CVTLT — BF1CVTLT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
1	11	BF1CVTLT, BF2CVTLT — BF2CVTLT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)

SVE2 FP8 downconverts

The encodings in this section are decoded from SVE floating-point unary operations - unpredicated.

31–24	23–22	21–19	18–16	15–12	11–10	9–6	5	4–0
01100101	00	001	010	0011	opc	Zn	0	Zd

opc	Instruction Details	Feature
00	FCVTN	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
01	FCVTNB	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
10	BFCVTN	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
11	FCVTNT (unpredicated)	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)

SVE floating-point reciprocal estimate (unpredicated)

The encodings in this section are decoded from SVE floating-point unary operations - unpredicated.

31–24	23–22	21–17	16	15–12	11–10	9–5	4–0
01100101	size	00111	op	0011	00	Zn	Zd

op	Instruction Details	Feature
0	FRECPE	FEAT_SVE || FEAT_SME
1	FRSQRTE	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE floating-point compare - with zero

SVE floating-point compare - with zero

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21–19	18	17–16	15–13	12–0
0 1 1 0 0 1 0 1		0 1 0	op0		0 0 1	

op0	Instruction details
0	SVE floating-point compare with zero
1	UNALLOCATED

SVE floating-point compare with zero

The encodings in this section are decoded from SVE floating-point compare - with zero.

31–24	23–22	21–19	18	17	16	15–13	12–10	9–5	4	3–0
0 1 1 0 0 1 0 1	size	0 1 0	0	eq	lt	0 0 1	Pg	Zn	ne	Pd

eq	lt	ne	Instruction Details	Feature
0	0	0	FCM<cc> (zero) — Greater than or equal	FEAT_SVE || FEAT_SME
0	0	1	FCM<cc> (zero) — Greater than	FEAT_SVE || FEAT_SME
0	1	0	FCM<cc> (zero) — Less than	FEAT_SVE || FEAT_SME
0	1	1	FCM<cc> (zero) — Less than or equal	FEAT_SVE || FEAT_SME
1	x	1	UNALLOCATED	-
1	0	0	FCM<cc> (zero) — Equal	FEAT_SVE || FEAT_SME
1	1	0	FCM<cc> (zero) — Not equal	FEAT_SVE || FEAT_SME

SVE floating-point accumulating reduction

SVE floating-point accumulating reduction

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21–19	18	17–16	15–13	12–0
0 1 1 0 0 1 0 1		0 1 1	op0		0 0 1	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0	SVE floating-point serial reduction (predicated)
1	UNALLOCATED

SVE floating-point serial reduction (predicated)

The encodings in this section are decoded from SVE floating-point accumulating reduction.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4		0
0	1	1	0	0	1	0	1	size		0	1	1	0	opc		0	0	1	Pg			Zm			Vdn		

opc	Instruction Details	Feature
00	FADDA	FEAT_SVE
!=00	UNALLOCATED	-

SVE floating-point arithmetic - predicated

SVE floating-point arithmetic - predicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19	18		16	15		13	12		10	9		6	5		0
0	1	1	0	0	1	0	1			0	op0					1	0	0	op1			op2					

op0	op1	op2	Instruction details	Feature
0x	xxx	xxxx	SVE floating-point arithmetic (predicated)	-
10	000	xxxx	FTMAD	FEAT_SVE
10	!=000	xxxx	UNALLOCATED	-
11	xxx	0000	SVE floating-point arithmetic with immediate (predicated)	-
11	xxx	!=0000	UNALLOCATED	-

SVE floating-point arithmetic (predicated)

The encodings in this section are decoded from SVE floating-point arithmetic - predicated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20	19			16	15			13	12			10	9					5	4					0
0	1	1	0	0	1	0	1	size		0	0	opc				1	0	0		Pg				Zm						Zdn					

size	opc	Instruction Details	Feature
xx	0011	FSUBR (vectors)	FEAT_SVE || FEAT_SME
xx	1000	FABD	FEAT_SVE || FEAT_SME
xx	1010	FMULX	FEAT_SVE || FEAT_SME
xx	1011	UNALLOCATED	-
xx	1100	FDIVR	FEAT_SVE || FEAT_SME
xx	1101	FDIV	FEAT_SVE || FEAT_SME
xx	1110	FAMAX	(FEAT_SVE2 && FEAT_FAMINMAX) || (FEAT_SME2 && FEAT_FAMINMAX)
xx	1111	FAMIN	(FEAT_SVE2 && FEAT_FAMINMAX) || (FEAT_SME2 && FEAT_FAMINMAX)
00	0000	BFADD (predicated)	FEAT_SVE_B16B16
00	0001	BFSUB (predicated)	FEAT_SVE_B16B16
00	0010	BFMUL (vectors, predicated)	FEAT_SVE_B16B16
00	0100	BFMAXNM	FEAT_SVE_B16B16
00	0101	BFMINNM	FEAT_SVE_B16B16
00	0110	BFMAX	FEAT_SVE_B16B16
00	0111	BFMIN	FEAT_SVE_B16B16
00	1001	BFSCALE	FEAT_SVE_BFSCALE
!= 00	0000	FADD (vectors, predicated)	FEAT_SVE || FEAT_SME
!= 00	0001	FSUB (vectors, predicated)	FEAT_SVE || FEAT_SME
!= 00	0010	FMUL (vectors, predicated)	FEAT_SVE || FEAT_SME
!= 00	0100	FMAXNM (vectors)	FEAT_SVE || FEAT_SME
!= 00	0101	FMINNM (vectors)	FEAT_SVE || FEAT_SME
!= 00	0110	FMAX (vectors)	FEAT_SVE || FEAT_SME
!= 00	0111	FMIN (vectors)	FEAT_SVE || FEAT_SME
!= 00	1001	FSCALE	FEAT_SVE || FEAT_SME

SVE floating-point arithmetic with immediate (predicated)

The encodings in this section are decoded from SVE floating-point arithmetic - predicated.

31							24	23	22	21	20	19	18		16	15			13	12			10	9				6	5	4					0
0	1	1	0	0	1	0	1	size		0	1	1	opc			1	0	0		Pg				0	0	0	0		i1	Zdn					

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
000	FADD (immediate)	FEAT_SVE || FEAT_SME
001	FSUB (immediate)	FEAT_SVE || FEAT_SME
010	FMUL (immediate)	FEAT_SVE || FEAT_SME
011	FSUBR (immediate)	FEAT_SVE || FEAT_SME
100	FMAXNM (immediate)	FEAT_SVE || FEAT_SME
101	FMINNM (immediate)	FEAT_SVE || FEAT_SME
110	FMAX (immediate)	FEAT_SVE || FEAT_SME
111	FMIN (immediate)	FEAT_SVE || FEAT_SME

SVE floating-point unary operations - merging predicated

SVE floating-point unary operations - merging predicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20		18	17	16	15			13	12											0
0	1	1	0	0	1	0	1			0	op0					1	0	1													

op0	Instruction details
00x	SVE floating-point round to integral value
010	SVE floating-point convert precision
011	SVE floating-point unary operations
10x	SVE integer convert to floating-point
11x	SVE floating-point convert to integer

SVE floating-point round to integral value

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

31							24	23	22	21			19	18			16	15			13	12			10	9					5	4					0
0	1	1	0	0	1	0	1	size		0	0	0		opc				1	0	1		Pg				Zn						Zd					

opc	Instruction Details	Feature
000	FRINT<r> — Nearest with ties to even, merging	FEAT_SVE || FEAT_SME
001	FRINT<r> — Toward plus infinity, merging	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
010	FRINT<r> — Toward minus infinity, merging	FEAT_SVE || FEAT_SME
011	FRINT<r> — Toward zero, merging	FEAT_SVE || FEAT_SME
100	FRINT<r> — Nearest with ties to away, merging	FEAT_SVE || FEAT_SME
101	UNALLOCATED	-
110	FRINT<r> — Current mode signalling inexact, merging	FEAT_SVE || FEAT_SME
111	FRINT<r> — Current mode, merging	FEAT_SVE || FEAT_SME

SVE floating-point convert precision

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

31							24	23	22	21	20			18	17	16	15			13	12			10	9					5	4						0
0	1	1	0	0	1	0	1	opc		0	0	1	0		opc2		1	0	1		Pg				Zn						Zd					

opc	opc2	Instruction Details	Feature
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTX	FEAT_SVE2 || FEAT_SME
01	xx	UNALLOCATED	-
10	00	FCVT — Single-precision to half-precision, merging	FEAT_SVE || FEAT_SME
10	01	FCVT — Half-precision to single-precision, merging	FEAT_SVE || FEAT_SME
10	10	BFCVT	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
11	00	FCVT — Double-precision to half-precision, merging	FEAT_SVE || FEAT_SME
11	01	FCVT — Half-precision to double-precision, merging	FEAT_SVE || FEAT_SME
11	10	FCVT — Double-precision to single-precision, merging	FEAT_SVE || FEAT_SME
11	11	FCVT — Single-precision to double-precision, merging	FEAT_SVE || FEAT_SME

SVE floating-point unary operations

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20		18	17	16	15		13	12		10	9		5	4		0
0	1	1	0	0	1	0	1	size		0	0	1	1	opc		1	0	1	Pg			Zn			Zd		

opc	Instruction Details	Feature
00	FRECPX	FEAT_SVE || FEAT_SME
01	FSQRT	FEAT_SVE || FEAT_SME
1x	UNALLOCATED	-

SVE integer convert to floating-point

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4		0
0	1	1	0	0	1	0	1	opc		0	1	0	opc2		U	1	0	1	Pg			Zn			Zd		

opc	opc2	U	Instruction Details	Feature
00	0x	0	FRINT32Z	FEAT_SVE2p2 || FEAT_SME2p2
00	0x	1	FRINT32X	FEAT_SVE2p2 || FEAT_SME2p2
00	1x	0	FRINT64Z	FEAT_SVE2p2 || FEAT_SME2p2
00	1x	1	FRINT64X	FEAT_SVE2p2 || FEAT_SME2p2
01	00	x	UNALLOCATED	-
01	01	0	SCVTF (predicated) — 16-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	01	1	UCVTF (predicated) — 16-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	10	0	SCVTF (predicated) — 32-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	10	1	UCVTF (predicated) — 32-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	11	0	SCVTF (predicated) — 64-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	11	1	UCVTF (predicated) — 64-bit to half-precision, merging	FEAT_SVE || FEAT_SME
10	10	0	SCVTF (predicated) — 32-bit to single-precision, merging	FEAT_SVE || FEAT_SME
10	10	1	UCVTF (predicated) — 32-bit to single-precision, merging	FEAT_SVE || FEAT_SME
10	!= 10	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	U	Instruction Details	Feature
11	00	0	SCVTF (predicated) — 32-bit to double-precision, merging	FEAT_SVE || FEAT_SME
11	00	1	UCVTF (predicated) — 32-bit to double-precision, merging	FEAT_SVE || FEAT_SME
11	01	x	UNALLOCATED	-
11	10	0	SCVTF (predicated) — 64-bit to single-precision, merging	FEAT_SVE || FEAT_SME
11	10	1	UCVTF (predicated) — 64-bit to single-precision, merging	FEAT_SVE || FEAT_SME
11	11	0	SCVTF (predicated) — 64-bit to double-precision, merging	FEAT_SVE || FEAT_SME
11	11	1	UCVTF (predicated) — 64-bit to double-precision, merging	FEAT_SVE || FEAT_SME

SVE floating-point convert to integer

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4		0
0	1	1	0	0	1	0	1	opc		0	1	1	opc2		U	1	0	1	Pg			Zn			Zd		

opc	opc2	U	Instruction Details	Feature
00	xx	0	FLOGB	FEAT_SVE2 || FEAT_SME
00	xx	1	UNALLOCATED	-
01	00	x	UNALLOCATED	-
01	01	0	FCVTZS — Half-precision to 16-bit, merging	FEAT_SVE || FEAT_SME
01	01	1	FCVTZU — Half-precision to 16-bit, merging	FEAT_SVE || FEAT_SME
01	10	0	FCVTZS — Half-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
01	10	1	FCVTZU — Half-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
01	11	0	FCVTZS — Half-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
01	11	1	FCVTZU — Half-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
10	10	0	FCVTZS — Single-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
10	10	1	FCVTZU — Single-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
10	!= 10	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	U	Instruction Details	Feature
11	00	0	FCVTZS — Double-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
11	00	1	FCVTZU — Double-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
11	01	x	UNALLOCATED	-
11	10	0	FCVTZS — Single-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
11	10	1	FCVTZU — Single-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
11	11	0	FCVTZS — Double-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
11	11	1	FCVTZU — Double-precision to 64-bit, merging	FEAT_SVE || FEAT_SME

SVE floating-point multiply-add

SVE floating-point multiply-add

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20	16	15	14	0
0	1	1	0	0	1	0	1			1			op0		

op0	Instruction details
0	SVE floating-point multiply-accumulate writing addend
1	SVE floating-point multiply-accumulate writing multiplicand

SVE floating-point multiply-accumulate writing addend

The encodings in this section are decoded from SVE floating-point multiply-add.

31							24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
0	1	1	0	0	1	0	1	size		1	Zm		0	opc		Pg		Zn		Zda	

size	opc	Instruction Details	Feature
00	00	BFMLA (vectors)	FEAT_SVE_B16B16
00	01	BFMLS (vectors)	FEAT_SVE_B16B16

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Feature
00	1x	UNALLOCATED	-
!=00	00	FMLA (vectors)	FEAT_SVE || FEAT_SME
!=00	01	FMLS (vectors)	FEAT_SVE || FEAT_SME
!=00	10	FNMLA	FEAT_SVE || FEAT_SME
!=00	11	FNMLS	FEAT_SVE || FEAT_SME

SVE floating-point multiply-accumulate writing multiplicand

The encodings in this section are decoded from SVE floating-point multiply-add.

31	24	23 22	21	20 16	15	14 13	12 10	9 5	4 0
0 1 1 0 0 1 0	1	size	1	Za	1	opc	Pg	Zm	Zdn

opc	Instruction Details	Feature
00	FMAD	FEAT_SVE || FEAT_SME
01	FMSB	FEAT_SVE || FEAT_SME
10	FNMAD	FEAT_SVE || FEAT_SME
11	FNMSB	FEAT_SVE || FEAT_SME

SVE Memory - 32-bit Gather and Unsized Contiguous

SVE Memory - 32-bit Gather and Unsized Contiguous

The encodings in this section are decoded from SVE encodings.

31	25	24 23	22 21	20 16	15 13	12 5	4	3 0
1 0 0 0 0 1 0		op0	op1		op2		op3	

op0	op1	op2	op3	Instruction details	Feature
00	x1	0xx	0	SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)	-
00	x1	0xx	1	UNALLOCATED	-
01	x1	0xx	x	SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)	-
10	x1	0xx	x	SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)	-
11	0x	000	0	LDR (predicate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
11	0x	000	1	UNALLOCATED	-
11	0x	010	x	LDR (vector)	FEAT_SVE || FEAT_SME
11	0x	0x1	x	UNALLOCATED	-
11	1x	0xx	0	SVE contiguous prefetch (scalar plus immediate)	-
11	1x	0xx	1	UNALLOCATED	-
xx	00	10x	x	SVE2 32-bit gather non-temporal load (vector plus scalar)	-
xx	00	110	0	SVE contiguous prefetch (scalar plus scalar)	-
xx	00	111	0	SVE 32-bit gather prefetch (vector plus immediate)	-
xx	00	11x	1	UNALLOCATED	-
xx	01	1xx	x	SVE 32-bit gather load (vector plus immediate)	-
xx	1x	1xx	x	SVE load and broadcast element	-
!= 11	x0	0xx	x	SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)	-

SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4	3			0
1	0	0	0	0	1	0	0	0	xs	1	Zm				0	msz		Pg			Rn				0	prfop			

msz	Instruction Details	Feature
00	PRFB (scalar plus vector)	FEAT_SVE
01	PRFH (scalar plus vector)	FEAT_SVE
10	PRFW (scalar plus vector)	FEAT_SVE
11	PRFD (scalar plus vector)	FEAT_SVE

SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4				0
1	0	0	0	0	1	0	0	1	xs	1	Zm				0	U	ff	Pg			Rn				Zt				

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	ff	Instruction Details	Feature
0	0	LD1SH (scalar plus vector)	FEAT_SVE
0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
1	0	LD1H (scalar plus vector)	FEAT_SVE
1	1	LDFF1H (scalar plus vector)	FEAT_SVE

SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4			0
1	0	0	0	0	1	0	1	0	xs	1	Zm				0	U	ff	Pg			Rn				Zt			

U	ff	Instruction Details	Feature
0	x	UNALLOCATED	-
1	0	LD1W (scalar plus vector)	FEAT_SVE
1	1	LDFF1W (scalar plus vector)	FEAT_SVE

SVE contiguous prefetch (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24	23	22	21				16	15	14	13	12		10	9			5	4	3			0
1	0	0	0	0	1	0	1	1	1	imm6					0	msz		Pg			Rn				0	prfop			

msz	Instruction Details	Feature
00	PRFB (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	PRFH (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	PRFW (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	PRFD (scalar plus immediate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 32-bit gather non-temporal load (vector plus scalar)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
1 0 0 0 0 1 0		msz		0	0	Rm		1	0	U	Pg		Zn		Zt	

msz	U	Instruction Details	Feature
00	0	LDNT1SB	FEAT_SVE2
00	1	LDNT1B (vector plus scalar)	FEAT_SVE2
01	0	LDNT1SH	FEAT_SVE2
01	1	LDNT1H (vector plus scalar)	FEAT_SVE2
10	0	UNALLOCATED	-
10	1	LDNT1W (vector plus scalar)	FEAT_SVE2
11	x	UNALLOCATED	-

SVE contiguous prefetch (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	0
1 0 0 0 0 1 0		msz		0	0	Rm		1 1 0		Pg		Rn		0	prfop	

msz	Rm	Instruction Details	Feature
xx	11111	UNALLOCATED	-
00	!= 11111	PRFB (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	!= 11111	PRFH (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	!= 11111	PRFW (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	!= 11111	PRFD (scalar plus scalar)	FEAT_SVE || FEAT_SME

SVE 32-bit gather prefetch (vector plus immediate)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	0
1 0 0 0 0 1 0		msz		0	0	imm5		1 1 1		Pg		Zn		0	prfop	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Feature
00	PRFB (vector plus immediate)	FEAT_SVE
01	PRFH (vector plus immediate)	FEAT_SVE
10	PRFW (vector plus immediate)	FEAT_SVE
11	PRFD (vector plus immediate)	FEAT_SVE

SVE 32-bit gather load (vector plus immediate)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1	0	0	0	0	1	0	msz	0 1	imm5	1	U	ff	Pg	Zn	Zt

msz	U	ff	Instruction Details	Feature
00	0	0	LD1SB (vector plus immediate)	FEAT_SVE
00	0	1	LDFF1SB (vector plus immediate)	FEAT_SVE
00	1	0	LD1B (vector plus immediate)	FEAT_SVE
00	1	1	LDFF1B (vector plus immediate)	FEAT_SVE
01	0	0	LD1SH (vector plus immediate)	FEAT_SVE
01	0	1	LDFF1SH (vector plus immediate)	FEAT_SVE
01	1	0	LD1H (vector plus immediate)	FEAT_SVE
01	1	1	LDFF1H (vector plus immediate)	FEAT_SVE
10	0	x	UNALLOCATED	-
10	1	0	LD1W (vector plus immediate)	FEAT_SVE
10	1	1	LDFF1W (vector plus immediate)	FEAT_SVE
11	x	x	UNALLOCATED	-

SVE load and broadcast element

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24 23	22	21 16	15	14 13	12 10	9 5	4 0
1	0	0	0	0	1	0	dtypeh	1	imm6	1	dtypel	Pg	Rn	Zt

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dtypeh	dtypel	Instruction Details	Feature
00	00	LD1RB — 8-bit element	FEAT_SVE || FEAT_SME
00	01	LD1RB — 16-bit element	FEAT_SVE || FEAT_SME
00	10	LD1RB — 32-bit element	FEAT_SVE || FEAT_SME
00	11	LD1RB — 64-bit element	FEAT_SVE || FEAT_SME
01	00	LD1RSW	FEAT_SVE || FEAT_SME
01	01	LD1RH — 16-bit element	FEAT_SVE || FEAT_SME
01	10	LD1RH — 32-bit element	FEAT_SVE || FEAT_SME
01	11	LD1RH — 64-bit element	FEAT_SVE || FEAT_SME
10	00	LD1RSH — 64-bit element	FEAT_SVE || FEAT_SME
10	01	LD1RSH — 32-bit element	FEAT_SVE || FEAT_SME
10	10	LD1RW — 32-bit element	FEAT_SVE || FEAT_SME
10	11	LD1RW — 64-bit element	FEAT_SVE || FEAT_SME
11	00	LD1RSB — 64-bit element	FEAT_SVE || FEAT_SME
11	01	LD1RSB — 32-bit element	FEAT_SVE || FEAT_SME
11	10	LD1RSB — 16-bit element	FEAT_SVE || FEAT_SME
11	11	LD1RD	FEAT_SVE || FEAT_SME

SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

The following constraints also apply to this encoding: opc != ’11’

opc	U	ff	Instruction Details	Feature
00	0	0	LD1SB (scalar plus vector)	FEAT_SVE
00	0	1	LDFF1SB (scalar plus vector)	FEAT_SVE
00	1	0	LD1B (scalar plus vector)	FEAT_SVE
00	1	1	LDFF1B (scalar plus vector)	FEAT_SVE
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	x	UNALLOCATED	-
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE Memory - Contiguous Load

SVE Memory - Contiguous Load

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20	19		16	15		13	12		0
1	0	1	0	0	1	0			op0		op1				op2					

op0	op1	op2	Instruction details
00	0	111	SVE contiguous non-temporal load (scalar plus immediate)
00	1	001	SVE contiguous load (quadwords, scalar plus immediate)
00	1	111	SVE load multiple structures (quadwords, scalar plus immediate)
00	x	100	SVE contiguous load (quadwords, scalar plus scalar)
00	x	110	SVE contiguous non-temporal load (scalar plus scalar)
01	x	100	SVE load multiple structures (quadwords, scalar plus scalar)
1x	x	100	UNALLOCATED
xx	0	001	SVE load and broadcast quadword (scalar plus immediate)
xx	0	101	SVE contiguous load (scalar plus immediate)
xx	1	101	SVE contiguous non-fault load (scalar plus immediate)
xx	x	000	SVE load and broadcast quadword (scalar plus scalar)
xx	x	010	SVE contiguous load (scalar plus scalar)
xx	x	011	SVE contiguous first-fault load (scalar plus scalar)
!= 00	0	111	SVE load multiple structures (scalar plus immediate)
!= 00	1	001	UNALLOCATED
!= 00	1	111	UNALLOCATED
!= 00	x	110	SVE load multiple structures (scalar plus scalar)

SVE contiguous non-temporal load (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31						25	24	23	22	21	20	19			16	15			13	12		10	9			5	4			0
1	0	1	0	0	1	0	msz		0	0	0	imm4				1	1	1		Pg			Rn				Zt			

msz	Instruction Details	Feature
00	LDNT1B (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
01	LDNT1H (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
10	LDNT1W (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
11	LDNT1D (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME

SVE contiguous load (quadwords, scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31						25	24	23	22	21	20	19			16	15			13	12		10	9			5	4			0
1	0	1	0	0	1	0	dtype		0	0	1	imm4				0	0	1		Pg			Rn				Zt			

dtype	Instruction Details	Feature
0x	UNALLOCATED	-
10	LD1W (scalar plus immediate, single register)	FEAT_SVE2p1
11	LD1D (scalar plus immediate, single register)	FEAT_SVE2p1

SVE load multiple structures (quadwords, scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31						25	24	23	22	21	20	19			16	15			13	12		10	9			5	4			0
1	0	1	0	0	1	0	num		0	0	1	imm4				1	1	1		Pg			Rn				Zt			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

num	Instruction Details	Feature
00	UNALLOCATED	-
01	LD2Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1
10	LD3Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1
11	LD4Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1

SVE contiguous load (quadwords, scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		dtype		0	0	Rm		1 0 0		Pg		Rn		Zt	

dtype	Rm	Instruction Details	Feature
0x	xxxxx	UNALLOCATED	-
1x	11111	UNALLOCATED	-
10	!= 11111	LD1W (scalar plus scalar, single register)	FEAT_SVE2p1
11	!= 11111	LD1D (scalar plus scalar, single register)	FEAT_SVE2p1

SVE contiguous non-temporal load (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		msz		0	0	Rm		1 1 0		Pg		Rn		Zt	

msz	Rm	Instruction Details	Feature
xx	11111	UNALLOCATED	-
00	!= 11111	LDNT1B (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
01	!= 11111	LDNT1H (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
10	!= 11111	LDNT1W (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
11	!= 11111	LDNT1D (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE load multiple structures (quadwords, scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		num		0	1	Rm		1 0 0		Pg		Rn		Zt	

num	Rm	Instruction Details	Feature
00	xxxxx	UNALLOCATED	-
01	!= 11111	LD2Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
10	!= 11111	LD3Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
11	!= 11111	LD4Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
!= 00	11111	UNALLOCATED	-

SVE load and broadcast quadword (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	23	22	21	20	19	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		msz		ssz		0	imm4		0 0 1		Pg		Rn		Zt	

msz	ssz	Instruction Details	Feature
xx	1x	UNALLOCATED	-
00	00	LD1RQB (scalar plus immediate)	FEAT_SVE || FEAT_SME
00	01	LD1ROB (scalar plus immediate)	FEAT_F64MM
01	00	LD1RQH (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	01	LD1ROH (scalar plus immediate)	FEAT_F64MM
10	00	LD1RQW (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	01	LD1ROW (scalar plus immediate)	FEAT_F64MM
11	00	LD1RQD (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	01	LD1ROD (scalar plus immediate)	FEAT_F64MM

SVE contiguous load (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	25	24	21	20	19	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		dtype		0	imm4		1 0 1		Pg		Rn		Zt	

dtype	Instruction Details	Feature
0000	LD1B (scalar plus immediate, single register) — 8-bit element	FEAT_SVE || FEAT_SME
0001	LD1B (scalar plus immediate, single register) — 16-bit element	FEAT_SVE || FEAT_SME
0010	LD1B (scalar plus immediate, single register) — 32-bit element	FEAT_SVE || FEAT_SME
0011	LD1B (scalar plus immediate, single register) — 64-bit element	FEAT_SVE || FEAT_SME
0100	LD1SW (scalar plus immediate)	FEAT_SVE || FEAT_SME
0101	LD1H (scalar plus immediate, single register) — 16-bit element	FEAT_SVE || FEAT_SME
0110	LD1H (scalar plus immediate, single register) — 32-bit element	FEAT_SVE || FEAT_SME
0111	LD1H (scalar plus immediate, single register) — 64-bit element	FEAT_SVE || FEAT_SME
1000	LD1SH (scalar plus immediate) — 64-bit element	FEAT_SVE || FEAT_SME
1001	LD1SH (scalar plus immediate) — 32-bit element	FEAT_SVE || FEAT_SME
1010	LD1W (scalar plus immediate, single register) — 32-bit element	FEAT_SVE || FEAT_SME
1011	LD1W (scalar plus immediate, single register) — 64-bit element	FEAT_SVE || FEAT_SME
1100	LD1SB (scalar plus immediate) — 64-bit element	FEAT_SVE || FEAT_SME
1101	LD1SB (scalar plus immediate) — 32-bit element	FEAT_SVE || FEAT_SME
1110	LD1SB (scalar plus immediate) — 16-bit element	FEAT_SVE || FEAT_SME
1111	LD1D (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME

SVE contiguous non-fault load (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	21	20	19	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		dtype		1	imm4		1 0 1		Pg		Rn		Zt	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dtype	Instruction Details	Feature
0000	LDNF1B — 8-bit element	FEAT_SVE
0001	LDNF1B — 16-bit element	FEAT_SVE
0010	LDNF1B — 32-bit element	FEAT_SVE
0011	LDNF1B — 64-bit element	FEAT_SVE
0100	LDNF1SW	FEAT_SVE
0101	LDNF1H — 16-bit element	FEAT_SVE
0110	LDNF1H — 32-bit element	FEAT_SVE
0111	LDNF1H — 64-bit element	FEAT_SVE
1000	LDNF1SH — 64-bit element	FEAT_SVE
1001	LDNF1SH — 32-bit element	FEAT_SVE
1010	LDNF1W — 32-bit element	FEAT_SVE
1011	LDNF1W — 64-bit element	FEAT_SVE
1100	LDNF1SB — 64-bit element	FEAT_SVE
1101	LDNF1SB — 32-bit element	FEAT_SVE
1110	LDNF1SB — 16-bit element	FEAT_SVE
1111	LDNF1D	FEAT_SVE

SVE load and broadcast quadword (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31 … 25	24 23	22 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 0 1 0 0 1 0	msz	ssz	Rm	0 0 0	Pg	Rn	Zt

msz	ssz	Rm	Instruction Details	Feature
xx	0x	11111	UNALLOCATED	-
xx	1x	xxxxx	UNALLOCATED	-
00	00	!= 11111	LD1RQB (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	01	!= 11111	LD1ROB (scalar plus scalar)	FEAT_F64MM
01	00	!= 11111	LD1RQH (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	01	!= 11111	LD1ROH (scalar plus scalar)	FEAT_F64MM
10	00	!= 11111	LD1RQW (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	01	!= 11111	LD1ROW (scalar plus scalar)	FEAT_F64MM
11	00	!= 11111	LD1RQD (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	01	!= 11111	LD1ROD (scalar plus scalar)	FEAT_F64MM

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE contiguous load (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31 … 25	24 … 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 0 1 0 0 1 0	dtype	Rm	0 1 0	Pg	Rn	Zt

dtype	Rm	Instruction Details	Feature
xxxx	11111	UNALLOCATED	-
0000	!= 11111	LD1B (scalar plus scalar, single register) — 8-bit element	FEAT_SVE || FEAT_SME
0001	!= 11111	LD1B (scalar plus scalar, single register) — 16-bit element	FEAT_SVE || FEAT_SME
0010	!= 11111	LD1B (scalar plus scalar, single register) — 32-bit element	FEAT_SVE || FEAT_SME
0011	!= 11111	LD1B (scalar plus scalar, single register) — 64-bit element	FEAT_SVE || FEAT_SME
0100	!= 11111	LD1SW (scalar plus scalar)	FEAT_SVE || FEAT_SME
0101	!= 11111	LD1H (scalar plus scalar, single register) — 16-bit element	FEAT_SVE || FEAT_SME
0110	!= 11111	LD1H (scalar plus scalar, single register) — 32-bit element	FEAT_SVE || FEAT_SME
0111	!= 11111	LD1H (scalar plus scalar, single register) — 64-bit element	FEAT_SVE || FEAT_SME
1000	!= 11111	LD1SH (scalar plus scalar) — 64-bit element	FEAT_SVE || FEAT_SME
1001	!= 11111	LD1SH (scalar plus scalar) — 32-bit element	FEAT_SVE || FEAT_SME
1010	!= 11111	LD1W (scalar plus scalar, single register) — 32-bit element	FEAT_SVE || FEAT_SME
1011	!= 11111	LD1W (scalar plus scalar, single register) — 64-bit element	FEAT_SVE || FEAT_SME
1100	!= 11111	LD1SB (scalar plus scalar) — 64-bit element	FEAT_SVE || FEAT_SME
1101	!= 11111	LD1SB (scalar plus scalar) — 32-bit element	FEAT_SVE || FEAT_SME
1110	!= 11111	LD1SB (scalar plus scalar) — 16-bit element	FEAT_SVE || FEAT_SME
1111	!= 11111	LD1D (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME

SVE contiguous first-fault load (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31 … 25	24 … 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 0 1 0 0 1 0	dtype	Rm	0 1 1	Pg	Rn	Zt

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dtype	Instruction Details	Feature
0000	LDFF1B (scalar plus scalar) — 8-bit element	FEAT_SVE
0001	LDFF1B (scalar plus scalar) — 16-bit element	FEAT_SVE
0010	LDFF1B (scalar plus scalar) — 32-bit element	FEAT_SVE
0011	LDFF1B (scalar plus scalar) — 64-bit element	FEAT_SVE
0100	LDFF1SW (scalar plus scalar)	FEAT_SVE
0101	LDFF1H (scalar plus scalar) — 16-bit element	FEAT_SVE
0110	LDFF1H (scalar plus scalar) — 32-bit element	FEAT_SVE
0111	LDFF1H (scalar plus scalar) — 64-bit element	FEAT_SVE
1000	LDFF1SH (scalar plus scalar) — 64-bit element	FEAT_SVE
1001	LDFF1SH (scalar plus scalar) — 32-bit element	FEAT_SVE
1010	LDFF1W (scalar plus scalar) — 32-bit element	FEAT_SVE
1011	LDFF1W (scalar plus scalar) — 64-bit element	FEAT_SVE
1100	LDFF1SB (scalar plus scalar) — 64-bit element	FEAT_SVE
1101	LDFF1SB (scalar plus scalar) — 32-bit element	FEAT_SVE
1110	LDFF1SB (scalar plus scalar) — 16-bit element	FEAT_SVE
1111	LDFF1D (scalar plus scalar)	FEAT_SVE

SVE load multiple structures (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31						25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1	0	1	0	0	1	0	msz	!= 00	0	imm4	1 1 1	Pg	Rn	Zt

opc

The following constraints also apply to this encoding: opc != ’00’

msz	opc	Instruction Details	Feature
00	01	LD2B (scalar plus immediate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	opc	Instruction Details	Feature
00	10	LD3B (scalar plus immediate)	FEAT_SVE || FEAT_SME
00	11	LD4B (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	01	LD2H (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	10	LD3H (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	11	LD4H (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	01	LD2W (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	10	LD3W (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	11	LD4W (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	01	LD2D (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	10	LD3D (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	11	LD4D (scalar plus immediate)	FEAT_SVE || FEAT_SME

SVE load multiple structures (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

The following constraints also apply to this encoding: opc != ’00’

msz	opc	Rm	Instruction Details	Feature
xx	!= 00	11111	UNALLOCATED	-
00	01	!= 11111	LD2B (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	10	!= 11111	LD3B (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	11	!= 11111	LD4B (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	01	!= 11111	LD2H (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	10	!= 11111	LD3H (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	11	!= 11111	LD4H (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	01	!= 11111	LD2W (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	10	!= 11111	LD3W (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	11	!= 11111	LD4W (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	01	!= 11111	LD2D (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	10	!= 11111	LD3D (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	11	!= 11111	LD4D (scalar plus scalar)	FEAT_SVE || FEAT_SME

SVE Memory - 64-bit Gather

SVE Memory - 64-bit Gather

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20		16	15		13	12		5	4	3		0
1	1	0	0	0	1	0	op0		op1					op2						op3			

op0	op1	op2	op3	Instruction details	Feature
00	00	101	x	LD1Q	FEAT_SVE2p1
00	01	0xx	1	UNALLOCATED	-
00	11	1xx	0	SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)	-
00	11	xxx	1	UNALLOCATED	-
00	x1	0xx	0	SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)	-
xx	00	111	0	SVE 64-bit gather prefetch (vector plus immediate)	-
xx	00	111	1	UNALLOCATED	-
xx	00	1x0	x	SVE2 64-bit gather non-temporal load (vector plus scalar)	-
xx	01	1xx	x	SVE 64-bit gather load (vector plus immediate)	-
xx	10	1xx	x	SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)	-
xx	x0	0xx	x	SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)	-
!=00	00	101	x	UNALLOCATED	-
!=00	11	1xx	x	SVE 64-bit gather load (scalar plus 64-bit scaled offsets)	-
!=00	x1	0xx	x	SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)	-

SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20		16	15	14	13	12		10	9		5	4	3		0
1	1	0	0	0	1	0	0	0	1	1		Zm		1	msz			Pg			Rn		0		prfop	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Feature
00	PRFB (scalar plus vector)	FEAT_SVE
01	PRFH (scalar plus vector)	FEAT_SVE
10	PRFW (scalar plus vector)	FEAT_SVE
11	PRFD (scalar plus vector)	FEAT_SVE

SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15	14		13	12			10	9				5	4	3				0
1	1	0	0	0	1	0	0	0	xs	1	Zm				0	msz			Pg				Rn					0	prfop				

msz	Instruction Details	Feature
00	PRFB (scalar plus vector)	FEAT_SVE
01	PRFH (scalar plus vector)	FEAT_SVE
10	PRFW (scalar plus vector)	FEAT_SVE
11	PRFD (scalar plus vector)	FEAT_SVE

SVE 64-bit gather prefetch (vector plus immediate)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15			13	12			10	9				5	4	3				0
1	1	0	0	0	1	0	msz		0	0	imm5				1	1		1	Pg				Zn					0	prfop				

msz	Instruction Details	Feature
00	PRFB (vector plus immediate)	FEAT_SVE
01	PRFH (vector plus immediate)	FEAT_SVE
10	PRFW (vector plus immediate)	FEAT_SVE
11	PRFD (vector plus immediate)	FEAT_SVE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 64-bit gather non-temporal load (vector plus scalar)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20				16	15	14	13	12			10	9				5	4				0
1	1	0	0	0	1	0	msz		0	0	Rm					1	U	0	Pg				Zn					Zt				

msz	U	Instruction Details	Feature
00	0	LDNT1SB	FEAT_SVE2
00	1	LDNT1B (vector plus scalar)	FEAT_SVE2
01	0	LDNT1SH	FEAT_SVE2
01	1	LDNT1H (vector plus scalar)	FEAT_SVE2
10	0	LDNT1SW	FEAT_SVE2
10	1	LDNT1W (vector plus scalar)	FEAT_SVE2
11	0	UNALLOCATED	-
11	1	LDNT1D (vector plus scalar)	FEAT_SVE2

SVE 64-bit gather load (vector plus immediate)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20				16	15	14	13	12			10	9				5	4				0
1	1	0	0	0	1	0	msz		0	1	imm5					1	U	ff	Pg				Zn					Zt				

msz	U	ff	Instruction Details	Feature
00	0	0	LD1SB (vector plus immediate)	FEAT_SVE
00	0	1	LDFF1SB (vector plus immediate)	FEAT_SVE
00	1	0	LD1B (vector plus immediate)	FEAT_SVE
00	1	1	LDFF1B (vector plus immediate)	FEAT_SVE
01	0	0	LD1SH (vector plus immediate)	FEAT_SVE
01	0	1	LDFF1SH (vector plus immediate)	FEAT_SVE
01	1	0	LD1H (vector plus immediate)	FEAT_SVE
01	1	1	LDFF1H (vector plus immediate)	FEAT_SVE
10	0	0	LD1SW (vector plus immediate)	FEAT_SVE
10	0	1	LDFF1SW (vector plus immediate)	FEAT_SVE
10	1	0	LD1W (vector plus immediate)	FEAT_SVE
10	1	1	LDFF1W (vector plus immediate)	FEAT_SVE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	U	ff	Instruction Details	Feature
11	0	x	UNALLOCATED	-
11	1	0	LD1D (vector plus immediate)	FEAT_SVE
11	1	1	LDFF1D (vector plus immediate)	FEAT_SVE

SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4			0
1	1	0	0	0	1	0	msz		1	0	Zm				1	U	ff	Pg			Rn				Zt			

msz	U	ff	Instruction Details	Feature
00	0	0	LD1SB (scalar plus vector)	FEAT_SVE
00	0	1	LDFF1SB (scalar plus vector)	FEAT_SVE
00	1	0	LD1B (scalar plus vector)	FEAT_SVE
00	1	1	LDFF1B (scalar plus vector)	FEAT_SVE
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	0	LD1SW (scalar plus vector)	FEAT_SVE
10	0	1	LDFF1SW (scalar plus vector)	FEAT_SVE
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE
11	0	x	UNALLOCATED	-
11	1	0	LD1D (scalar plus vector)	FEAT_SVE
11	1	1	LDFF1D (scalar plus vector)	FEAT_SVE

SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4			0
1	1	0	0	0	1	0	msz		xs	0	Zm				0	U	ff	Pg			Rn				Zt			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	U	ff	Instruction Details	Feature
00	0	0	LD1SB (scalar plus vector)	FEAT_SVE
00	0	1	LDFF1SB (scalar plus vector)	FEAT_SVE
00	1	0	LD1B (scalar plus vector)	FEAT_SVE
00	1	1	LDFF1B (scalar plus vector)	FEAT_SVE
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	0	LD1SW (scalar plus vector)	FEAT_SVE
10	0	1	LDFF1SW (scalar plus vector)	FEAT_SVE
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE
11	0	x	UNALLOCATED	-
11	1	0	LD1D (scalar plus vector)	FEAT_SVE
11	1	1	LDFF1D (scalar plus vector)	FEAT_SVE

SVE 64-bit gather load (scalar plus 64-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

The following constraints also apply to this encoding: opc != ’00’

opc	U	ff	Instruction Details	Feature
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	0	LD1SW (scalar plus vector)	FEAT_SVE
10	0	1	LDFF1SW (scalar plus vector)	FEAT_SVE
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE
11	0	x	UNALLOCATED	-
11	1	0	LD1D (scalar plus vector)	FEAT_SVE
11	1	1	LDFF1D (scalar plus vector)	FEAT_SVE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4			0
1	1	0	0	0	1	0	!= 00		xs	1	Zm				0	U	ff	Pg			Rn				Zt			
							opc																					

The following constraints also apply to this encoding: opc != ’00’

opc	U	ff	Instruction Details	Feature
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	0	LD1SW (scalar plus vector)	FEAT_SVE
10	0	1	LDFF1SW (scalar plus vector)	FEAT_SVE
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE
11	0	x	UNALLOCATED	-
11	1	0	LD1D (scalar plus vector)	FEAT_SVE
11	1	1	LDFF1D (scalar plus vector)	FEAT_SVE

SVE Memory - Non-temporal and Quadword Scatter Store

SVE Memory - Non-temporal and Quadword Scatter Store

The encodings in this section are decoded from SVE encodings.

31						25	24		22	21	20			16	15		13	12										0
1	1	1	0	0	1	0	op0			op1					0	0	1											

op0	op1	Instruction details	Feature
000	1	ST1Q	FEAT_SVE2p1
xx0	0	SVE2 64-bit scatter non-temporal store (vector plus scalar)	-
xx1	0	SVE2 32-bit scatter non-temporal store (vector plus scalar)	-
!= 000	1	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 64-bit scatter non-temporal store (vector plus scalar)

The encodings in this section are decoded from SVE Memory - Non-temporal and Quadword Scatter Store.

31–25	24–23	22	21	20–16	15–13	12–10	9–5	4–0
1 1 1 0 0 1 0	msz	0	0	Rm	0 0 1	Pg	Zn	Zt

msz	Instruction Details	Feature
00	STNT1B (vector plus scalar)	FEAT_SVE2
01	STNT1H (vector plus scalar)	FEAT_SVE2
10	STNT1W (vector plus scalar)	FEAT_SVE2
11	STNT1D (vector plus scalar)	FEAT_SVE2

SVE2 32-bit scatter non-temporal store (vector plus scalar)

The encodings in this section are decoded from SVE Memory - Non-temporal and Quadword Scatter Store.

31–25	24–23	22	21	20–16	15–13	12–10	9–5	4–0
1 1 1 0 0 1 0	msz	1	0	Rm	0 0 1	Pg	Zn	Zt

msz	Instruction Details	Feature
00	STNT1B (vector plus scalar)	FEAT_SVE2
01	STNT1H (vector plus scalar)	FEAT_SVE2
10	STNT1W (vector plus scalar)	FEAT_SVE2
11	UNALLOCATED	-

SVE Memory - Non-temporal and Multi-register Contiguous Store

SVE Memory - Non-temporal and Multi-register Contiguous Store

The encodings in this section are decoded from SVE encodings.

31–25	24–23	22–21	20–16	15–13	12–0
1 1 1 0 0 1 0		op0		0 1 1	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
00	SVE contiguous non-temporal store (scalar plus scalar)
!= 00	SVE store multiple structures (scalar plus scalar)

SVE contiguous non-temporal store (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Contiguous Store.

31–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1 0 0 1 0	msz	0 0	Rm	0 1 1	Pg	Rn	Zt

msz	Rm	Instruction Details	Feature
xx	11111	UNALLOCATED	-
00	!= 11111	STNT1B (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
01	!= 11111	STNT1H (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
10	!= 11111	STNT1W (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
11	!= 11111	STNT1D (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME

SVE store multiple structures (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Contiguous Store.

31–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1 0 0 1 0	msz	!= 00 (opc)	Rm	0 1 1	Pg	Rn	Zt

The following constraints also apply to this encoding: opc != ’00’

msz	opc	Rm	Instruction Details	Feature
xx	!= 00	11111	UNALLOCATED	-
00	01	!= 11111	ST2B (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	10	!= 11111	ST3B (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	11	!= 11111	ST4B (scalar plus scalar)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	opc	Rm	Instruction Details	Feature
01	01	!= 11111	ST2H (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	10	!= 11111	ST3H (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	11	!= 11111	ST4H (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	01	!= 11111	ST2W (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	10	!= 11111	ST3W (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	11	!= 11111	ST4W (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	01	!= 11111	ST2D (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	10	!= 11111	ST3D (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	11	!= 11111	ST4D (scalar plus scalar)	FEAT_SVE || FEAT_SME

SVE Memory - Contiguous Store and Unsized Contiguous

SVE Memory - Contiguous Store and Unsized Contiguous

The encodings in this section are decoded from SVE encodings.

31						25	24	22	21	20	19	16	15	14	13	12	5	4	3	0
1	1	1	0	0	1	0	op0		op1				0	op2	0			op3		

op0	op1	op2	op3	Instruction details	Feature
0xx	00	0	x	SVE store multiple structures (quadwords, scalar plus immediate)	-
0xx	01	0	x	UNALLOCATED	-
0xx	1x	0	x	SVE store multiple structures (quadwords, scalar plus scalar)	-
10x	xx	0	x	UNALLOCATED	-
110	xx	0	0	STR (predicate)	FEAT_SVE || FEAT_SME
110	xx	0	1	UNALLOCATED	-
110	xx	1	x	STR (vector)	FEAT_SVE || FEAT_SME
111	xx	0	x	UNALLOCATED	-
!= 110	xx	1	x	SVE contiguous store (scalar plus scalar)	-

SVE store multiple structures (quadwords, scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Store and Unsized Contiguous.

31							24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	0
1	1	1	0	0	1	0	0	num		0	0	imm4		0	0	0	Pg		Rn		Zt	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

num	Instruction Details	Feature
00	UNALLOCATED	-
01	ST2Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1
10	ST3Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1
11	ST4Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1

SVE store multiple structures (quadwords, scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Store and Unsized Contiguous.

31	24	23 22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 1 0 0 1 0 0		num	1	Rm	0	0	0	Pg	Rn	Zt

num	Rm	Instruction Details	Feature
00	xxxxx	UNALLOCATED	-
01	!= 11111	ST2Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
10	!= 11111	ST3Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
11	!= 11111	ST4Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
!= 00	11111	UNALLOCATED	-

SVE contiguous store (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Store and Unsized Contiguous.

31	25	24 22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 1 0 0 1 0		!= 110	o2	Rm	0	1	0	Pg	Rn	Zt
		opc								

The following constraints also apply to this encoding: opc != ’110’

opc	o2	Rm	Instruction Details	Feature
xx1	x	11111	UNALLOCATED	-
0x0	x	11111	UNALLOCATED	-
00x	x	!= 11111	ST1B (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
01x	x	!= 11111	ST1H (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
100	0	11111	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	Rm	Instruction Details	Feature
100	0	!= 11111	ST1W (scalar plus scalar, single register) — SVE2	FEAT_SVE2p1
100	1	xxxxx	UNALLOCATED	-
101	x	!= 11111	ST1W (scalar plus scalar, single register) — SVE	FEAT_SVE || FEAT_SME
111	0	!= 11111	ST1D (scalar plus scalar, single register) — SVE2	FEAT_SVE2p1
111	1	!= 11111	ST1D (scalar plus scalar, single register) — SVE	FEAT_SVE || FEAT_SME

SVE Memory - Scatter

SVE Memory - Scatter

The encodings in this section are decoded from SVE encodings.

31	25	24	23	22	21	20	16	15	13	12	0
1 1 1 0 0 1 0				op0				1 0 1			

op0	Instruction details
00	SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)
01	SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)
10	SVE 64-bit scatter store (vector plus immediate)
11	SVE 32-bit scatter store (vector plus immediate)

SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	0
1 1 1 0 0 1 0		msz		0	0	Zm		1 0 1		Pg		Rn		Zt	

msz	Instruction Details	Feature
00	ST1B (scalar plus vector)	FEAT_SVE
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Feature
11	ST1D (scalar plus vector)	FEAT_SVE

SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter.

31						25	24	23	22	21	20				16	15			12			10	9				5	4				0
1	1	1	0	0	1	0	msz		0	1	Zm					1	0	1	Pg				Rn					Zt				

msz	Instruction Details	Feature
00	UNALLOCATED	-
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	ST1D (scalar plus vector)	FEAT_SVE

SVE 64-bit scatter store (vector plus immediate)

The encodings in this section are decoded from SVE Memory - Scatter.

31						25	24	23	22	21	20				16	15			12			10	9				5	4				0
1	1	1	0	0	1	0	msz		1	0	imm5					1	0	1	Pg				Zn					Zt				

msz	Instruction Details	Feature
00	ST1B (vector plus immediate)	FEAT_SVE
01	ST1H (vector plus immediate)	FEAT_SVE
10	ST1W (vector plus immediate)	FEAT_SVE
11	ST1D (vector plus immediate)	FEAT_SVE

SVE 32-bit scatter store (vector plus immediate)

The encodings in this section are decoded from SVE Memory - Scatter.

31						25	24	23	22	21	20				16	15			12			10	9				5	4				0
1	1	1	0	0	1	0	msz		1	1	imm5					1	0	1	Pg				Zn					Zt				

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Feature
00	ST1B (vector plus immediate)	FEAT_SVE
01	ST1H (vector plus immediate)	FEAT_SVE
10	ST1W (vector plus immediate)	FEAT_SVE
11	UNALLOCATED	-

SVE Memory - Contiguous Store with Immediate Offset

SVE Memory - Contiguous Store with Immediate Offset

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20	19			16	15			13	12															0
1	1	1	0	0	1	0			op0		op1					1	1	1																	

op0	op1	Instruction details
00	1	SVE contiguous non-temporal store (scalar plus immediate)
xx	0	SVE contiguous store (scalar plus immediate)
!=00	1	SVE store multiple structures (scalar plus immediate)

SVE contiguous non-temporal store (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Store with Immediate Offset.

31						25	24	23	22	21	20	19			16	15			13	12			10	9					5	4					0
1	1	1	0	0	1	0	msz		0	0	1	imm4				1	1	1		Pg				Rn						Zt					

msz	Instruction Details	Feature
00	STNT1B (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
01	STNT1H (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
10	STNT1W (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Feature
11	STNT1D (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME

SVE contiguous store (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Store with Immediate Offset.

31	25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1 0 0 1 0		msz	opc	0	imm4	1 1 1	Pg	Rn	Zt

msz	opc	Instruction Details	Feature
00	xx	ST1B (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
01	xx	ST1H (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
10	00	ST1W (scalar plus immediate, single register) — SVE2	FEAT_SVE2p1
10	01	UNALLOCATED	-
10	1x	ST1W (scalar plus immediate, single register) — SVE	FEAT_SVE || FEAT_SME
11	0x	UNALLOCATED	-
11	10	ST1D (scalar plus immediate, single register) — SVE2	FEAT_SVE2p1
11	11	ST1D (scalar plus immediate, single register) — SVE	FEAT_SVE || FEAT_SME

SVE store multiple structures (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Store with Immediate Offset.

31	25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1 0 0 1 0		msz	!= 00	1	imm4	1 1 1	Pg	Rn	Zt
			opc						

The following constraints also apply to this encoding: opc != ’00’

msz	opc	Instruction Details	Feature
00	01	ST2B (scalar plus immediate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	opc	Instruction Details	Feature
00	10	ST3B (scalar plus immediate)	FEAT_SVE || FEAT_SME
00	11	ST4B (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	01	ST2H (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	10	ST3H (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	11	ST4H (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	01	ST2W (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	10	ST3W (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	11	ST4W (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	01	ST2D (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	10	ST3D (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	11	ST4D (scalar plus immediate)	FEAT_SVE || FEAT_SME

SVE Memory - Scatter with Optional Sign Extend

SVE Memory - Scatter with Optional Sign Extend

The encodings in this section are decoded from SVE encodings.

31	25	24	23	22	21	20	16	15	14	13	12	0
1 1 1 0 0 1 0				op0				1		0		

op0	Instruction details
00	SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)
01	SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)
10	SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)
11	SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend.

31	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
1 1 1 0 0 1 0		msz		0	0	Zm		1	xs	0	Pg		Rn		Zt	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Feature
00	ST1B (scalar plus vector)	FEAT_SVE
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	ST1D (scalar plus vector)	FEAT_SVE

SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend.

31						25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1	1	1	0	0	1	0	msz	0 1	Zm	1	xs	0	Pg	Rn	Zt

msz	Instruction Details	Feature
00	UNALLOCATED	-
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	ST1D (scalar plus vector)	FEAT_SVE

SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend.

31						25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1	1	1	0	0	1	0	msz	1 0	Zm	1	xs	0	Pg	Rn	Zt

msz	Instruction Details	Feature
00	ST1B (scalar plus vector)	FEAT_SVE
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend.

31						25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
1	1	1	0	0	1	0	msz		1	1	Zm		1	xs	0	Pg		Rn		Zt	

msz	Instruction Details	Feature
00	UNALLOCATED	-
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	UNALLOCATED	-

SVE integer add/subtract vectors (unpredicated)

The encodings in this section are decoded from SVE encodings.

31		29	28				24	23	22	21	20	16	15		13	12	10	9	5	4	0
0	0	0	0	0	1	0	0	size		1	Zm		0	0	0	opc		Zn		Zd	

size	opc	Instruction Details	Feature
xx	000	ADD (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	001	SUB (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	100	SQADD (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	101	UQADD (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	110	SQSUB (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	111	UQSUB (vectors, unpredicated)	FEAT_SVE || FEAT_SME
11	010	ADDPT (unpredicated)	FEAT_SVE && FEAT_CPA
11	011	SUBPT (unpredicated)	FEAT_SVE && FEAT_CPA
!= 11	01x	UNALLOCATED	-

SVE address generation

The encodings in this section are decoded from SVE encodings.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–29	28–24	23–22	21	20–16	15–12	11–10	9–5	4–0
0 0 0	0 0 1 0 0	opc	1	Zm	1 0 1 0	msz	Zn	Zd

opc	Instruction Details	Feature
00	ADR — Unpacked 32-bit signed offsets	FEAT_SVE
01	ADR — Unpacked 32-bit unsigned offsets	FEAT_SVE
1x	ADR — Packed offsets	FEAT_SVE

SVE table lookup (three sources)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–11	10	9–5	4–0
0 0 0	0 0 1 0 1	size	1	Zm	0 0 1 0 1	op	Zn	Zd

op	Instruction Details	Feature
0	TBL	FEAT_SVE2 || FEAT_SME
1	TBX	FEAT_SVE2 || FEAT_SME

SVE permute vector elements

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0 1	size	1	Zm	0 1 1	opc	Zn	Zd

opc	Instruction Details	Feature
000	ZIP1, ZIP2 (vectors) — Low halves	FEAT_SVE || FEAT_SME
001	ZIP1, ZIP2 (vectors) — High halves	FEAT_SVE || FEAT_SME
010	UZP1, UZP2 (vectors) — Even	FEAT_SVE || FEAT_SME
011	UZP1, UZP2 (vectors) — Odd	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
100	TRN1, TRN2 (vectors) — Even	FEAT_SVE || FEAT_SME
101	TRN1, TRN2 (vectors) — Odd	FEAT_SVE || FEAT_SME
11x	UNALLOCATED	-

SVE integer compare with unsigned immediate

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–14	13	12–10	9–5	4	3–0
0 0 1	0 0 1 0 0		1	imm7	lt	Pg	Zn	ne	Pd

lt	ne	Instruction Details	Feature
0	0	CMP<cc> (immediate) — Higher or same	FEAT_SVE || FEAT_SME
0	1	CMP<cc> (immediate) — Higher	FEAT_SVE || FEAT_SME
1	0	CMP<cc> (immediate) — Lower	FEAT_SVE || FEAT_SME
1	1	CMP<cc> (immediate) — Lower or same	FEAT_SVE || FEAT_SME

SVE predicate logical operations

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23	22	21–20	19–16	15–14	13–10	9	8–5	4	3–0
0 0 1	0 0 1 0 1	op	S	0 0	Pm	0 1	Pg	o2	Pn	o3	Pd

op	S	o2	o3	Instruction Details	Feature
0	0	0	0	AND (predicates)	FEAT_SVE || FEAT_SME
0	0	0	1	BIC (predicates)	FEAT_SVE || FEAT_SME
0	0	1	0	EOR (predicates)	FEAT_SVE || FEAT_SME
0	0	1	1	SEL (predicates)	FEAT_SVE || FEAT_SME
0	1	0	0	ANDS	FEAT_SVE || FEAT_SME
0	1	0	1	BICS	FEAT_SVE || FEAT_SME
0	1	1	0	EORS	FEAT_SVE || FEAT_SME
0	1	1	1	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	o2	o3	Instruction Details	Feature
1	0	0	0	ORR (predicates)	FEAT_SVE || FEAT_SME
1	0	0	1	ORN (predicates)	FEAT_SVE || FEAT_SME
1	0	1	0	NOR	FEAT_SVE || FEAT_SME
1	0	1	1	NAND	FEAT_SVE || FEAT_SME
1	1	0	0	ORRS	FEAT_SVE || FEAT_SME
1	1	0	1	ORNS	FEAT_SVE || FEAT_SME
1	1	1	0	NORS	FEAT_SVE || FEAT_SME
1	1	1	1	NANDS	FEAT_SVE || FEAT_SME

SVE integer compare with signed immediate

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23 22	21	20 16	15	14	13	12 10	9 5	4	3 0
0 0 1	0 0 1 0 1	size	0	imm5	op	0	o2	Pg	Zn	ne	Pd

op	o2	ne	Instruction Details	Feature
0	0	0	CMP<cc> (immediate) — Greater than or equal	FEAT_SVE || FEAT_SME
0	0	1	CMP<cc> (immediate) — Greater than	FEAT_SVE || FEAT_SME
0	1	0	CMP<cc> (immediate) — Less than	FEAT_SVE || FEAT_SME
0	1	1	CMP<cc> (immediate) — Less than or equal	FEAT_SVE || FEAT_SME
1	0	0	CMP<cc> (immediate) — Equal	FEAT_SVE || FEAT_SME
1	0	1	CMP<cc> (immediate) — Not equal	FEAT_SVE || FEAT_SME
1	1	x	UNALLOCATED	-

SVE broadcast predicate element

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23	22	21	20 18	17 16	15 14	13 10	9	8 5	4	3 0
0 0 1	0 0 1 0 1	i1	tszh	1	tszl	Rv	0 1	Pn	S	Pm	0	Pd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	Instruction Details	Feature
0	PSEL	FEAT_SME || FEAT_SVE2p1
1	UNALLOCATED	-

SVE two-way dot product

The encodings in this section are decoded from SVE encodings.

31–29	28–21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0 0 0 0 0	Zm	1 1 0 0 1	U	Zn	Zda

U	Instruction Details	Feature
0	SDOT (2-way, vectors)	FEAT_SME2 || FEAT_SVE2p1
1	UDOT (2-way, vectors)	FEAT_SME2 || FEAT_SVE2p1

SVE two-way dot product (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0 0 1 0 0	opc	1 1 0 0 1	U	Zn	Zda

U	Instruction Details	Feature
0	SDOT (2-way, indexed)	FEAT_SME2 || FEAT_SVE2p1
1	UDOT (2-way, indexed)	FEAT_SME2 || FEAT_SVE2p1

SVE integer clamp

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0 0	size	0	Zm	1 1 0 0 0	U	Zn	Zd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Feature
0	SCLAMP	FEAT_SME || FEAT_SVE2p1
1	UCLAMP	FEAT_SME || FEAT_SVE2p1

SVE2 multiply-add (checked pointer)

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20				16	15			12	11	10	9				5	4				0
0	1	0	0	0	1	0	0	opc		0	Zm					1	1	0	1	o2	0	Zn					Zda				

opc	o2	Instruction Details	Feature
11	0	MLAPT	FEAT_SVE && FEAT_CPA
11	1	MADPT	FEAT_SVE && FEAT_CPA
!= 11	x	UNALLOCATED	-

SVE permute vector elements (quadwords)

The encodings in this section are decoded from SVE encodings.

31		29	28				24	23	22	21	20				16	15		13	12		10	9				5	4				0
0	1	0	0	0	1	0	0	size		0	Zm					1	1	1	opc			Zn					Zd				

opc	Instruction Details	Feature
000	ZIPQ1	FEAT_SVE2p1 || FEAT_SME2p1
001	ZIPQ2	FEAT_SVE2p1 || FEAT_SME2p1
010	UZPQ1	FEAT_SVE2p1 || FEAT_SME2p1
011	UZPQ2	FEAT_SVE2p1 || FEAT_SME2p1
10x	UNALLOCATED	-
110	TBLQ	FEAT_SVE2p1 || FEAT_SME2p1
111	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 character match

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–13	12–10	9–5	4	3–0
0 1 0	0 0 1 0 1	size	1	Zm	1 0 0	Pg	Zn	op	Pd

op	Instruction Details	Feature
0	MATCH	FEAT_SVE2
1	NMATCH	FEAT_SVE2

SVE2 FP8 matrix multiply-accumulate

The encodings in this section are decoded from SVE encodings.

31–29	28–23	22	21	20–16	15–10	9–5	4–0
0 1 1	0 0 1 0 0 0	op	1	Zm	1 1 1 0 0 0	Zn	Zda

op	Instruction Details	Feature
0	FMMLA (widening, FP8 to FP32)	FEAT_SVE2 && FEAT_F8F32MM
1	FMMLA (widening, FP8 to FP16)	FEAT_SVE2 && FEAT_F8F16MM

SVE2 FP8 multiply-add long (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23	22–21	20–19	18–16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0 0	T	0 1	i4h	Zm	0 1 0 1	i4l	Zn	Zda

T	Instruction Details	Feature
0	FMLALB (indexed, FP8 to FP16)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
1	FMLALT (indexed, FP8 to FP16)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE floating-point convert (top, predicated)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
011	00100	opc	0000	opc2	101	Pg	Zn	Zd

opc	opc2	Instruction Details	Feature
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTXNT	FEAT_SVE2p2 || FEAT_SME2p2
01	xx	UNALLOCATED	-
10	00	FCVTNT (predicated) — Single-precision to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	01	FCVTLT — Half-precision to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	10	BFCVTNT	FEAT_SVE2p2 || FEAT_SME2p2
11	0x	UNALLOCATED	-
11	10	FCVTNT (predicated) — Double-precision to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	11	FCVTLT — Single-precision to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

SVE floating-point convert precision odd elements

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
011	00100	opc	0010	opc2	101	Pg	Zn	Zd

opc	opc2	Instruction Details	Feature
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTXNT	FEAT_SVE2 || FEAT_SME
01	xx	UNALLOCATED	-
10	00	FCVTNT (predicated) — Single-precision to half-precision, merging	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	Instruction Details	Feature
10	01	FCVTLT — Half-precision to single-precision, merging	FEAT_SVE2 || FEAT_SME
10	10	BFCVTNT	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
11	0x	UNALLOCATED	-
11	10	FCVTNT (predicated) — Double-precision to single-precision, merging	FEAT_SVE2 || FEAT_SME
11	11	FCVTLT — Single-precision to double-precision, merging	FEAT_SVE2 || FEAT_SME

SVE2 floating-point pairwise operations

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23 22	21 19	18 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0 0		0 1 0	opc	1 0 0	Pg	Zm	Zdn

opc	Instruction Details	Feature
000	FADDP	FEAT_SVE2 || FEAT_SME
001	UNALLOCATED	-
01x	UNALLOCATED	-
100	FMAXNMP	FEAT_SVE2 || FEAT_SME
101	FMINNMP	FEAT_SVE2 || FEAT_SME
110	FMAXP	FEAT_SVE2 || FEAT_SME
111	FMINP	FEAT_SVE2 || FEAT_SME

SVE floating-point recursive reduction (quadwords)

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23 22	21 19	18 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0 0	size	0 1 0	opc	1 0 1	Pg	Zn	Vd

opc	Instruction Details	Feature
000	FADDQV	FEAT_SVE2p1 || FEAT_SME2p1
001	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
01x	UNALLOCATED	-
100	FMAXNMQV	FEAT_SVE2p1 || FEAT_SME2p1
101	FMINNMQV	FEAT_SVE2p1 || FEAT_SME2p1
110	FMAXQV	FEAT_SVE2p1 || FEAT_SME2p1
111	FMINQV	FEAT_SVE2p1 || FEAT_SME2p1

SVE floating-point multiply-add (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0 0	size	1	opc	0 0 0 0	o2	op	Zn	Zda

size	o2	op	Instruction Details	Feature
0x	0	0	FMLA (indexed) — Half-precision	FEAT_SVE || FEAT_SME
0x	0	1	FMLS (indexed) — Half-precision	FEAT_SVE || FEAT_SME
0x	1	0	BFMLA (indexed)	FEAT_SVE_B16B16
0x	1	1	BFMLS (indexed)	FEAT_SVE_B16B16
1x	1	x	UNALLOCATED	-
10	0	0	FMLA (indexed) — Single-precision	FEAT_SVE || FEAT_SME
10	0	1	FMLS (indexed) — Single-precision	FEAT_SVE || FEAT_SME
11	0	0	FMLA (indexed) — Double-precision	FEAT_SVE || FEAT_SME
11	0	1	FMLS (indexed) — Double-precision	FEAT_SVE || FEAT_SME

SVE floating-point complex multiply-add (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0 0	size	1	opc	0 0 0 1	rot	Zn	Zda

size	Instruction Details	Feature
0x	UNALLOCATED	-
10	FCMLA (indexed) — Half-precision	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Instruction Details	Feature
11	FCMLA (indexed) — Single-precision	FEAT_SVE || FEAT_SME

SVE FP clamp

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23 22	21	20 16	15 10	9 5	4 0
0 1 1	0 0 1 0 0	size	1	Zm	0 0 1 0 0 1	Zn	Zd

size	Instruction Details	Feature
00	BFCLAMP	FEAT_SVE_B16B16
!= 00	FCLAMP	FEAT_SME2 || FEAT_SVE2p1

SVE floating-point multiply (indexed)

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23 22	21	20 16	15 12	11	10	9 5	4 0
0 1 1	0 0 1 0 0	size	1	opc	0 0 1 0	o2	0	Zn	Zd

size	o2	Instruction Details	Feature
0x	0	FMUL (indexed) — Half-precision	FEAT_SVE || FEAT_SME
0x	1	BFMUL (indexed)	FEAT_SVE_B16B16
1x	1	UNALLOCATED	-
10	0	FMUL (indexed) — Single-precision	FEAT_SVE || FEAT_SME
11	0	FMUL (indexed) — Double-precision	FEAT_SVE || FEAT_SME

SVE2 FP8 multiply-add long long (indexed)

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23 22	21	20 19	18 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0 0	TT	1	i4h	Zm	1 1 0 0	i4l	Zn	Zda

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TT	Instruction Details	Feature
00	FMLALLBB (indexed)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
01	FMLALLBT (indexed)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
10	FMLALLTB (indexed)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
11	FMLALLTT (indexed)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)

SVE floating-point matrix multiply accumulate

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–10	9–5	4–0
011	00100	opc	1	Zm	111001	Zn	Zda

opc	Instruction Details	Feature
00	FMMLA (widening, FP16 to FP32)	FEAT_SVE_F16F32MM
01	BFMMLA (widening)	FEAT_SVE && FEAT_BF16
10	FMMLA (non-widening) — 32-bit element	FEAT_F32MM
11	FMMLA (non-widening) — 64-bit element	FEAT_F64MM

SVE floating-point recursive reduction

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
011	00101	size	000	opc	001	Pg	Zn	Vd

opc	Instruction Details	Feature
000	FADDV	FEAT_SVE || FEAT_SME
001	UNALLOCATED	-
01x	UNALLOCATED	-
100	FMAXNMV	FEAT_SVE || FEAT_SME
101	FMINNMV	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
110	FMAXV	FEAT_SVE || FEAT_SME
111	FMINV	FEAT_SVE || FEAT_SME

SVE floating-point arithmetic (unpredicated)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0 1	size	0	Zm	0 0 0	opc	Zn	Zd

size	opc	Instruction Details	Feature
xx	011	FTSMUL	FEAT_SVE
xx	10x	UNALLOCATED	-
xx	110	FRECPS	FEAT_SVE || FEAT_SME
xx	111	FRSQRTS	FEAT_SVE || FEAT_SME
00	000	BFADD (unpredicated)	FEAT_SVE_B16B16
00	001	BFSUB (unpredicated)	FEAT_SVE_B16B16
00	010	BFMUL (vectors, unpredicated)	FEAT_SVE_B16B16
!= 00	000	FADD (vectors, unpredicated)	FEAT_SVE || FEAT_SME
!= 00	001	FSUB (vectors, unpredicated)	FEAT_SVE || FEAT_SME
!= 00	010	FMUL (vectors, unpredicated)	FEAT_SVE || FEAT_SME

SVE floating-point compare vectors

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15	14	13	12–10	9–5	4	3–0
0 1 1	0 0 1 0 1	size	0	Zm	op	1	o2	Pg	Zn	o3	Pd

op	o2	o3	Instruction Details	Feature
0	0	0	FCM<cc> (vectors) — Greater than or equal	FEAT_SVE || FEAT_SME
0	0	1	FCM<cc> (vectors) — Greater than	FEAT_SVE || FEAT_SME
0	1	0	FCM<cc> (vectors) — Equal	FEAT_SVE || FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	o2	o3	Instruction Details	Feature
0	1	1	FCM<cc> (vectors) — Not equal	FEAT_SVE || FEAT_SME
1	0	0	FCM<cc> (vectors) — Unordered	FEAT_SVE || FEAT_SME
1	0	1	FAC<cc> — Greater than or equal	FEAT_SVE || FEAT_SME
1	1	0	UNALLOCATED	-
1	1	1	FAC<cc> — Greater than	FEAT_SVE || FEAT_SME

Data Processing -- Immediate

Data Processing – Immediate

The encodings in this section are decoded from A64 instruction set encoding.

31	30 29	28 26	25 22	21 0
	op0	1 0 0	op1	

op0	op1	Instruction details
11	111x	Data-processing (1 source immediate)
xx	00xx	PC-rel. addressing
xx	010x	Add/subtract (immediate)
xx	0110	Add/subtract (immediate, with tags)
xx	0111	Min/max (immediate)
xx	100x	Logical (immediate)
xx	101x	Move wide (immediate)
xx	110x	Bitfield
!=11	111x	Extract

Data-processing (1 source immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30	29	28					23	22	21	20 5	4 0
sf	1	1	1	0	0	1	1	1	opc		imm16	Rd

sf	opc	Rd	Instruction Details	Feature
0	xx	xxxxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	opc	Rd	Instruction Details	Feature
1	0x	!= 11111	UNALLOCATED	-
1	00	11111	AUTIASPPC	FEAT_PAuth_LR
1	01	11111	AUTIBSPPC	FEAT_PAuth_LR
1	1x	xxxxx	UNALLOCATED	-

PC-rel. addressing

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 24	23 5	4 0
op	immlo	1 0 0 0 0	immhi	Rd

op	Instruction Details
0	ADR
1	ADRP

Add/subtract (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30	29	28 23	22	21 10	9 5	4 0
sf	op	S	1 0 0 0 1 0	sh	imm12	Rn	Rd

sf	op	S	Instruction Details
0	0	0	ADD (immediate) — 32-bit
0	0	1	ADDS (immediate) — 32-bit
0	1	0	SUB (immediate) — 32-bit
0	1	1	SUBS (immediate) — 32-bit
1	0	0	ADD (immediate) — 64-bit
1	0	1	ADDS (immediate) — 64-bit
1	1	0	SUB (immediate) — 64-bit
1	1	1	SUBS (immediate) — 64-bit

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Add/subtract (immediate, with tags)

The encodings in this section are decoded from Data Processing – Immediate.

31	30	29	28 26	25 22	21 16	15 14	13 10	9 5	4 0
sf	op	S	1 0 0	0 1 1 0	imm6	op3	imm4	Rn	Rd

sf	op	S	Instruction Details	Feature
0	x	x	UNALLOCATED	-
1	x	1	UNALLOCATED	-
1	0	0	ADDG	FEAT_MTE
1	1	0	SUBG	FEAT_MTE

Min/max (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30	29	28 26	25 22	21 18	17 10	9 5	4 0
sf	op	S	1 0 0	0 1 1 1	opc	imm8	Rn	Rd

sf	op	S	opc	Instruction Details	Feature
x	0	0	01xx	UNALLOCATED	-
x	0	0	1xxx	UNALLOCATED	-
x	0	1	xxxx	UNALLOCATED	-
x	1	x	xxxx	UNALLOCATED	-
0	0	0	0000	SMAX (immediate) — 32-bit	FEAT_CSSC
0	0	0	0001	UMAX (immediate) — 32-bit	FEAT_CSSC
0	0	0	0010	SMIN (immediate) — 32-bit	FEAT_CSSC
0	0	0	0011	UMIN (immediate) — 32-bit	FEAT_CSSC
1	0	0	0000	SMAX (immediate) — 64-bit	FEAT_CSSC
1	0	0	0001	UMAX (immediate) — 64-bit	FEAT_CSSC
1	0	0	0010	SMIN (immediate) — 64-bit	FEAT_CSSC
1	0	0	0011	UMIN (immediate) — 64-bit	FEAT_CSSC

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Logical (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	opc	1 0 0 1 0 0	N	immr	imms	Rn	Rd

sf	opc	N	Instruction Details
0	xx	1	UNALLOCATED
0	00	0	AND (immediate) — 32-bit
0	01	0	ORR (immediate) — 32-bit
0	10	0	EOR (immediate) — 32-bit
0	11	0	ANDS (immediate) — 32-bit
1	00	x	AND (immediate) — 64-bit
1	01	x	ORR (immediate) — 64-bit
1	10	x	EOR (immediate) — 64-bit
1	11	x	ANDS (immediate) — 64-bit

Move wide (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 23	22 21	20 5	4 0
sf	opc	1 0 0 1 0 1	hw	imm16	Rd

sf	opc	hw	Instruction Details
x	01	0x	UNALLOCATED
0	xx	1x	UNALLOCATED
0	00	0x	MOVN — 32-bit
0	10	0x	MOVZ — 32-bit
0	11	0x	MOVK — 32-bit
1	00	xx	MOVN — 64-bit
1	01	1x	UNALLOCATED
1	10	xx	MOVZ — 64-bit
1	11	xx	MOVK — 64-bit

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bitfield

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	opc	1 0 0 1 1 0	N	immr	imms	Rn	Rd

sf	opc	N	Instruction Details
0	xx	1	UNALLOCATED
0	00	0	SBFM — 32-bit
0	01	0	BFM — 32-bit
0	10	0	UBFM — 32-bit
0	11	0	UNALLOCATED
1	xx	0	UNALLOCATED
1	00	1	SBFM — 64-bit
1	01	1	BFM — 64-bit
1	10	1	UBFM — 64-bit
1	11	1	UNALLOCATED

Extract

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 23	22	21	20 16	15 10	9 5	4 0
sf	!= 11	1 0 0 1 1 1	N	o0	Rm	imms	Rn	Rd

op21

The following constraints also apply to this encoding: op21 != ’11’

sf	op21	N	o0	imms	Instruction Details
x	00	x	1	xxxxxx	UNALLOCATED
x	01	x	x	xxxxxx	UNALLOCATED
x	10	x	x	xxxxxx	UNALLOCATED
0	00	0	0	0xxxxx	EXTR — 32-bit
0	00	0	0	1xxxxx	UNALLOCATED
0	00	1	0	xxxxxx	UNALLOCATED
1	00	0	0	xxxxxx	UNALLOCATED
1	00	1	0	xxxxxx	EXTR — 64-bit

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Branches, Exception Generating and System instructions

Branches, Exception Generating and System instructions

The encodings in this section are decoded from A64 instruction set encoding.

31–29	28–26	25–12	11–5	4–0
op0	1 0 1	op1		op2

op0	op1	op2	Instruction details
010	00xxxxxxxxxxxx	xxxxx	Conditional branch (immediate)
010	01xxxxxxxxxxxx	xxxxx	Miscellaneous branch (immediate)
011	00xxxxxxxx1xxx	xxxxx	Compare bytes/halfwords in registers and branch
01x	1xxxxxxxxxxxxx	xxxxx	UNALLOCATED
110	00xxxxxxxxxxxx	xxxxx	Exception generation
110	010000000x00xx	xxxxx	UNALLOCATED
110	010000001000xx	xxxxx	UNALLOCATED
110	01000000110000	xxxxx	UNALLOCATED
110	01000000110001	xxxxx	System instructions with register argument
110	01000000110010	11111	Hints
110	01000000110010	!= 11111	UNALLOCATED
110	01000000110011	xxxxx	Barriers
110	01000001xx00xx	xxxxx	UNALLOCATED
110	0100000xxx0100	xxxxx	PSTATE
110	0100000xxx0101	xxxxx	UNALLOCATED
110	0100000xxx011x	xxxxx	UNALLOCATED
110	0100000xxx1xxx	xxxxx	UNALLOCATED
110	0100100xxxxxxx	xxxxx	UNALLOCATED
110	0100x01xxxxxxx	xxxxx	System instructions
110	0100x1xxxxxxxx	xxxxx	System register move
110	0101x00xxxxxxx	xxxxx	UNALLOCATED
110	0101x01xxxxxxx	xxxxx	System pair instructions
110	0101x1xxxxxxxx	xxxxx	System register pair move
110	011xxxxxxxxxxx	xxxxx	UNALLOCATED
110	1xxxxxxxxxxxxx	xxxxx	Unconditional branch (register)
111	00xxxxxxxx1xxx	xxxxx	UNALLOCATED
111	1xxxxxxxxxxxxx	xxxxx	UNALLOCATED
x00	xxxxxxxxxxxxxx	xxxxx	Unconditional branch (immediate)
x01	0xxxxxxxxxxxxx	xxxxx	Compare and branch (immediate)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
x01	1xxxxxxxxxxxxx	xxxxx	Test and branch (immediate)
x11	00xxxxxxxx00xx	xxxxx	Compare registers and branch
x11	00xxxxxxxx01xx	xxxxx	UNALLOCATED
x11	01xxxxxxxxx0xx	xxxxx	Compare register with immediate and branch
x11	01xxxxxxxxx1xx	xxxxx	UNALLOCATED

Conditional branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31 29	28 24	23 5	4	3 0
0 1 0	1 0 1 0 0	imm19	o0	cond

o0	Instruction Details	Feature
0	B.cond	-
1	BC.cond	FEAT_HBC

Miscellaneous branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31 29	28 24	23 21	20 5	4 0
0 1 0	1 0 1 0 1	opc	imm16	op2

opc	op2	Instruction Details			Feature
00x	!=11111	UNALLOCATED			-
000	11111	RETAASPPC, RETAASPPC	RETABSPPC	—	FEAT_PAuth_LR
001	11111	RETAASPPC, RETABSPPC	RETABSPPC	—	FEAT_PAuth_LR
01x	xxxxx	UNALLOCATED			-
1xx	xxxxx	UNALLOCATED			-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Compare bytes/halfwords in registers and branch

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1 0 0	cc	Rm	1	H	imm9	Rt

cc	H	Instruction Details	Feature
000	0	CBB<cc> — Greater than	FEAT_CMPBR
000	1	CBH<cc> — Greater than	FEAT_CMPBR
001	0	CBB<cc> — Greater than or equal	FEAT_CMPBR
001	1	CBH<cc> — Greater than or equal	FEAT_CMPBR
010	0	CBB<cc> — Higher	FEAT_CMPBR
010	1	CBH<cc> — Higher	FEAT_CMPBR
011	0	CBB<cc> — Higher or same	FEAT_CMPBR
011	1	CBH<cc> — Higher or same	FEAT_CMPBR
10x	x	UNALLOCATED	-
110	0	CBB<cc> — Equal	FEAT_CMPBR
110	1	CBH<cc> — Equal	FEAT_CMPBR
111	0	CBB<cc> — Not equal	FEAT_CMPBR
111	1	CBH<cc> — Not equal	FEAT_CMPBR

Exception generation

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–24	23–21	20–5	4–2	1–0
1 1 0	1 0 1 0 0	opc	imm16	op2	LL

opc	op2	LL	Instruction Details
xxx	!= 000	xx	UNALLOCATED
x11	000	xx	UNALLOCATED
000	000	00	UNALLOCATED
000	000	01	SVC
000	000	10	HVC
000	000	11	SMC

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	op2	LL	Instruction Details
001	000	00	BRK
001	000	!= 00	UNALLOCATED
010	000	00	HLT
010	000	!= 00	UNALLOCATED
1x0	000	xx	UNALLOCATED
101	000	00	UNALLOCATED
101	000	01	DCPS1
101	000	10	DCPS2
101	000	11	DCPS3

System instructions with register argument

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 0 1	CRm	op2	Rt

CRm	op2	Instruction Details	Feature
0000	000	WFET	FEAT_WFxT
0000	001	WFIT	FEAT_WFxT
0000	01x	UNALLOCATED	-
0000	1xx	UNALLOCATED	-
!= 0000	xxx	UNALLOCATED	-

Hints

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	CRm	op2	1 1 1 1 1

CRm	op2	Instruction Details	Feature
xxxx	xxx	HINT	-
0000	000	NOP	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRm	op2	Instruction Details	Feature
0000	001	YIELD	-
0000	010	WFE	-
0000	011	WFI	-
0000	100	SEV	-
0000	101	SEVL	-
0000	110	DGH	FEAT_DGH
0000	111	XPACD, XPACI, XPACLRI	FEAT_PAuth
0001	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA1716	FEAT_PAuth
0001	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB1716	FEAT_PAuth
0001	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA1716	FEAT_PAuth
0001	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB1716	FEAT_PAuth
0010	000	ESB	FEAT_RAS
0010	001	PSB	FEAT_SPE
0010	010	TSB	FEAT_TRF
0010	011	GCSB	FEAT_GCS
0010	100	CSDB	-
0010	110	CLRBHB	FEAT_CLRBHB
0011	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIAZ	FEAT_PAuth
0011	001	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIASP	FEAT_PAuth
0011	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBZ	FEAT_PAuth
0011	011	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBSP	FEAT_PAuth
0011	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIAZ	FEAT_PAuth
0011	101	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIASP	FEAT_PAuth
0011	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBZ	FEAT_PAuth
0011	111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBSP	FEAT_PAuth
0100	xx0	BTI	FEAT_BTI
0100	111	PACM	FEAT_PAuth_LR
0101	000	CHKFEAT	FEAT_CHK
0110	00x	STSHH	FEAT_PCDPHINT

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Barriers

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	op2	Rt

CRm	op2	Rt	Instruction Details	Feature
xxxx	xxx	!= 11111	UNALLOCATED	-
xxxx	000	11111	UNALLOCATED	-
xxxx	010	11111	CLREX	-
xxxx	100	11111	DSB — Memory barrier	-
xxxx	101	11111	DMB	-
xxxx	110	11111	ISB	-
xxxx	111	11111	SB	FEAT_SB
xx0x	0x1	11111	UNALLOCATED	-
xx10	001	11111	DSB — Memory nXS barrier	FEAT_XS
xx10	011	11111	UNALLOCATED	-
xx11	0x1	11111	UNALLOCATED	-

PSTATE

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1 0 1 0 0 0 0 0	op1	0 1 0 0	CRm	op2	Rt

op1:op2	Rt	Instruction Details	Feature
xxxxxx	0xxxx	UNALLOCATED	-
xxxxxx	10xxx	UNALLOCATED	-
xxxxxx	110xx	UNALLOCATED	-
xxxxxx	1110x	UNALLOCATED	-
xxxxxx	11110	UNALLOCATED	-
000000	11111	CFINV	FEAT_FlagM
000001	11111	XAFLAG	FEAT_FlagM2
000010	11111	AXFLAG	FEAT_FlagM2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1:op2	Rt	Instruction Details	Feature
`!=00000x`	`11111`	MSR (immediate)	-

System instructions

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	29	28	22	21	20	19	18	16	15	12	11	8	7	5	4	0
1 1 0		1 0 1 0 1 0 0		L	0	1	op1		CRn		CRm		op2		Rt	

L	Instruction Details
0	SYS
1	SYSL

System register move

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	29	28	22	21	20	19	18	16	15	12	11	8	7	5	4	0
1 1 0		1 0 1 0 1 0 0		L	1	o0	op1		CRn		CRm		op2		Rt	

L	Instruction Details
0	MSR (register)
1	MRS

System pair instructions

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	29	28	22	21	20	19	18	16	15	12	11	8	7	5	4	0
1 1 0		1 0 1 0 1 0 1		L	0	1	op1		CRn		CRm		op2		Rt	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	Instruction Details	Feature
0	SYSP	FEAT_SYSINSTR128
1	UNALLOCATED	-

System register pair move

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1 0 1 0 1	L	1	o0	op1	CRn	CRm	op2	Rt

L	Instruction Details	Feature
0	MSRR	FEAT_SYSREG128
1	MRRS	FEAT_SYSREG128

Unconditional branch (register)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–25	24–21	20–16	15–10	9–5	4–0
1 1 0	1 0 1 1	opc	op2	op3	Rn	op4

opc	op2	op3	Rn	op4	Instruction Details	Feature
xxxx	11111	0001xx	xxxxx	xxxxx	UNALLOCATED	-
xxxx	11111	001xxx	xxxxx	xxxxx	UNALLOCATED	-
xxxx	11111	01xxxx	xxxxx	xxxxx	UNALLOCATED	-
xxxx	11111	1xxxxx	xxxxx	xxxxx	UNALLOCATED	-
xxxx	!= 11111	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
00x0	11111	000000	xxxxx	!= 00000	UNALLOCATED	-
00x0	11111	000001	xxxxx	xxxxx	UNALLOCATED	-
000x	11111	00001x	xxxxx	!= 11111	UNALLOCATED	-
0000	11111	000000	xxxxx	00000	BR	-
0000	11111	000010	xxxxx	11111	BRAA, BRAAZ, BRAB, BRABZ — Key A, zero modifier	FEAT_PAuth

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	op2	op3	Rn	op4	Instruction Details	Feature
0000	11111	000011	xxxxx	11111	BRAA, BRAAZ, BRAB, BRABZ — Key B, zero modifier	FEAT_PAuth
0001	11111	000000	xxxxx	00000	BLR	-
0001	11111	000000	xxxxx	!= 00000	UNALLOCATED	-
0001	11111	000001	xxxxx	xxxxx	UNALLOCATED	-
0001	11111	000010	xxxxx	11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key A, zero modifier	FEAT_PAuth
0001	11111	000011	xxxxx	11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key B, zero modifier	FEAT_PAuth
0010	11111	000000	xxxxx	00000	RET	-
0010	11111	00001x	!= 11111	xxxxx	UNALLOCATED	-
0010	11111	000010	11111	11111	RETAA, RETAB — RETAA	FEAT_PAuth
0010	11111	000010	11111	!= 11111	RETAASPPCR, RETABSPPCR — RETAASPPCR	FEAT_PAuth_LR
0010	11111	000011	11111	11111	RETAA, RETAB — RETAB	FEAT_PAuth
0010	11111	000011	11111	!= 11111	RETAASPPCR, RETABSPPCR — RETABSPPCR	FEAT_PAuth_LR
0011	11111	0000xx	xxxxx	xxxxx	UNALLOCATED	-
010x	11111	0000xx	!= 11111	xxxxx	UNALLOCATED	-
010x	11111	000000	11111	!= 00000	UNALLOCATED	-
010x	11111	000001	11111	xxxxx	UNALLOCATED	-
010x	11111	00001x	11111	xxxx0	UNALLOCATED	-
0100	11111	000000	11111	00000	ERET	-
0100	11111	00001x	11111	0xxx1	UNALLOCATED	-
0100	11111	00001x	11111	10xx1	UNALLOCATED	-
0100	11111	00001x	11111	110x1	UNALLOCATED	-
0100	11111	00001x	11111	11101	UNALLOCATED	-
0100	11111	000010	11111	11111	ERETAA, ERETAB — ERETAA	FEAT_PAuth
0100	11111	000011	11111	11111	ERETAA, ERETAB — ERETAB	FEAT_PAuth
0101	11111	000000	11111	00000	DRPS	-
0101	11111	00001x	11111	xxxx1	UNALLOCATED	-
011x	11111	0000xx	xxxxx	xxxxx	UNALLOCATED	-
100x	11111	00000x	xxxxx	xxxxx	UNALLOCATED	-
1000	11111	000010	xxxxx	xxxxx	BRAA, BRAAZ, BRAB, BRABZ — Key A, register modifier	FEAT_PAuth
1000	11111	000011	xxxxx	xxxxx	BRAA, BRAAZ, BRAB, BRABZ — Key B, register modifier	FEAT_PAuth
1001	11111	000010	xxxxx	xxxxx	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key A, register modifier	FEAT_PAuth
1001	11111	000011	xxxxx	xxxxx	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key B, register modifier	FEAT_PAuth
101x	11111	0000xx	xxxxx	xxxxx	UNALLOCATED	-
11xx	11111	0000xx	xxxxx	xxxxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Unconditional branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30				26	25 … 0
op	0	0	1	0	1	imm26

op	Instruction Details
0	B
1	BL

Compare and branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30					25	24	23 … 5	4 … 0
sf	0	1	1	0	1	0	op	imm19	Rt

sf	op	Instruction Details
0	0	CBZ — 32-bit
0	1	CBNZ — 32-bit
1	0	CBZ — 64-bit
1	1	CBNZ — 64-bit

Test and branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30					25	24	23 … 19	18 … 5	4 … 0
b5	0	1	1	0	1	1	op	b40	imm14	Rt

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details
0	TBZ
1	TBNZ

Compare registers and branch

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30						24	23	21	20	16	15	14	13	5	4	0
sf	1	1	1	0	1	0	0	cc		Rm		0	0	imm9		Rt	

sf	cc	Instruction Details	Feature
x	10x	UNALLOCATED	-
0	000	CB<cc> (register) — 32-bit greater than	FEAT_CMPBR
0	001	CB<cc> (register) — 32-bit greater than or equal	FEAT_CMPBR
0	010	CB<cc> (register) — 32-bit higher	FEAT_CMPBR
0	011	CB<cc> (register) — 32-bit higher or same	FEAT_CMPBR
0	110	CB<cc> (register) — 32-bit equal	FEAT_CMPBR
0	111	CB<cc> (register) — 32-bit not equal	FEAT_CMPBR
1	000	CB<cc> (register) — 64-bit greater than	FEAT_CMPBR
1	001	CB<cc> (register) — 64-bit greater than or equal	FEAT_CMPBR
1	010	CB<cc> (register) — 64-bit higher	FEAT_CMPBR
1	011	CB<cc> (register) — 64-bit higher or same	FEAT_CMPBR
1	110	CB<cc> (register) — 64-bit equal	FEAT_CMPBR
1	111	CB<cc> (register) — 64-bit not equal	FEAT_CMPBR

Compare register with immediate and branch

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30						24	23	21	20	15	14	13	5	4	0
sf	1	1	1	0	1	0	1	cc		imm6		0	imm9		Rt	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	cc	Instruction Details	Feature
x	10x	UNALLOCATED	-
0	000	CB<cc> (immediate) — 32-bit greater than	FEAT_CMPBR
0	001	CB<cc> (immediate) — 32-bit less than	FEAT_CMPBR
0	010	CB<cc> (immediate) — 32-bit higher	FEAT_CMPBR
0	011	CB<cc> (immediate) — 32-bit lower	FEAT_CMPBR
0	110	CB<cc> (immediate) — 32-bit equal	FEAT_CMPBR
0	111	CB<cc> (immediate) — 32-bit not equal	FEAT_CMPBR
1	000	CB<cc> (immediate) — 64-bit greater than	FEAT_CMPBR
1	001	CB<cc> (immediate) — 64-bit less than	FEAT_CMPBR
1	010	CB<cc> (immediate) — 64-bit higher	FEAT_CMPBR
1	011	CB<cc> (immediate) — 64-bit lower	FEAT_CMPBR
1	110	CB<cc> (immediate) — 64-bit equal	FEAT_CMPBR
1	111	CB<cc> (immediate) — 64-bit not equal	FEAT_CMPBR

Data Processing -- Register

Data Processing – Register

The encodings in this section are decoded from A64 instruction set encoding.

op0	op1	op2	op3	Instruction details
0	1	0110	xxxxxx	Data-processing (2 source)
1	1	0110	xxxxxx	Data-processing (1 source)
x	0	0xxx	xxxxxx	Logical (shifted register)
x	0	1xx0	xxxxxx	Add/subtract (shifted register)
x	0	1xx1	xxxxxx	Add/subtract (extended register)
x	1	0000	000000	Add/subtract (with carry)
x	1	0000	001xxx	Add/subtract (checked pointer)
x	1	0000	011xxx	UNALLOCATED
x	1	0000	100000	UNALLOCATED
x	1	0000	1x1xxx	UNALLOCATED
x	1	0000	x00001	Rotate right into flags
x	1	0000	x0010x	UNALLOCATED
x	1	0000	x10x0x	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details
x	1	0000	xx0010	Evaluate into flags
x	1	0000	xx0011	UNALLOCATED
x	1	0000	xx011x	UNALLOCATED
x	1	0010	xxxx0x	Conditional compare (register)
x	1	0010	xxxx1x	Conditional compare (immediate)
x	1	0100	xxxxxx	Conditional select
x	1	0xx1	xxxxxx	UNALLOCATED
x	1	1xxx	xxxxxx	Data-processing (3 source)

Data-processing (2 source)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	10	9	5	4	0
sf	0	S	1	1	0 1	0	1 1 0	Rm		opcode		Rn		Rd	

sf	S	opcode	Instruction Details	Feature
x	x	1xxxxx	UNALLOCATED	-
x	0	00011x	UNALLOCATED	-
x	1	00001x	UNALLOCATED	-
x	1	0001xx	UNALLOCATED	-
x	1	001xxx	UNALLOCATED	-
x	1	01xxxx	UNALLOCATED	-
0	x	00000x	UNALLOCATED	-
0	0	0x11xx	UNALLOCATED	-
0	0	000010	UDIV — 32-bit	-
0	0	000011	SDIV — 32-bit	-
0	0	00010x	UNALLOCATED	-
0	0	001000	LSLV — 32-bit	-
0	0	001001	LSRV — 32-bit	-
0	0	001010	ASRV — 32-bit	-
0	0	001011	RORV — 32-bit	-
0	0	010x11	UNALLOCATED	-
0	0	010000	CRC32B, CRC32H, CRC32W, CRC32X — CRC32B	FEAT_CRC32
0	0	010001	CRC32B, CRC32H, CRC32W, CRC32X — CRC32H	FEAT_CRC32

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	opcode	Instruction Details	Feature
0	0	010010	CRC32B, CRC32H, CRC32W, CRC32X — CRC32W	FEAT_CRC32
0	0	010100	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB	FEAT_CRC32
0	0	010101	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH	FEAT_CRC32
0	0	010110	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW	FEAT_CRC32
0	0	011000	SMAX (register) — 32-bit	FEAT_CSSC
0	0	011001	UMAX (register) — 32-bit	FEAT_CSSC
0	0	011010	SMIN (register) — 32-bit	FEAT_CSSC
0	0	011011	UMIN (register) — 32-bit	FEAT_CSSC
1	x	000001	UNALLOCATED	-
1	0	000000	SUBP	FEAT_MTE
1	0	000010	UDIV — 64-bit	-
1	0	000011	SDIV — 64-bit	-
1	0	000100	IRG	FEAT_MTE
1	0	000101	GMI	FEAT_MTE
1	0	001000	LSLV — 64-bit	-
1	0	001001	LSRV — 64-bit	-
1	0	001010	ASRV — 64-bit	-
1	0	001011	RORV — 64-bit	-
1	0	001100	PACGA	FEAT_PAuth
1	0	001101	UNALLOCATED	-
1	0	00111x	UNALLOCATED	-
1	0	010xx0	UNALLOCATED	-
1	0	010001	UNALLOCATED	-
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X — CRC32X	FEAT_CRC32
1	0	010101	UNALLOCATED	-
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX	FEAT_CRC32
1	0	011000	SMAX (register) — 64-bit	FEAT_CSSC
1	0	011001	UMAX (register) — 64-bit	FEAT_CSSC
1	0	011010	SMIN (register) — 64-bit	FEAT_CSSC
1	0	011011	UMIN (register) — 64-bit	FEAT_CSSC
1	0	0111xx	UNALLOCATED	-
1	1	000000	SUBPS	FEAT_MTE

Data-processing (1 source)

The encodings in this section are decoded from Data Processing – Register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29	28	27	25	24	21	20	16	15	10	9	5	4	0
sf	1	S	1	1	0 1	0	1 1 0	opcode2		opcode		Rn		Rd	

sf	S	opcode2	opcode	Rn	Rd	Instruction Details	Feature
x	0	00000	001001	xxxxx	xxxxx	UNALLOCATED	-
x	0	00000	00101x	xxxxx	xxxxx	UNALLOCATED	-
x	0	00000	0011xx	xxxxx	xxxxx	UNALLOCATED	-
x	0	00000	01xxxx	xxxxx	xxxxx	UNALLOCATED	-
x	0	00000	1xxxxx	xxxxx	xxxxx	UNALLOCATED	-
x	0	0001x	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
x	0	001xx	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
x	0	01xxx	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
x	0	1xxxx	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
x	1	xxxxx	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
0	0	00000	000000	xxxxx	xxxxx	RBIT — 32-bit	-
0	0	00000	000001	xxxxx	xxxxx	REV16 — 32-bit	-
0	0	00000	000010	xxxxx	xxxxx	REV — 32-bit	-
0	0	00000	000011	xxxxx	xxxxx	UNALLOCATED	-
0	0	00000	000100	xxxxx	xxxxx	CLZ — 32-bit	-
0	0	00000	000101	xxxxx	xxxxx	CLS — 32-bit	-
0	0	00000	000110	xxxxx	xxxxx	CTZ — 32-bit	FEAT_CSSC
0	0	00000	000111	xxxxx	xxxxx	CNT — 32-bit	FEAT_CSSC
0	0	00000	001000	xxxxx	xxxxx	ABS — 32-bit	FEAT_CSSC
0	0	00001	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
1	0	00000	000000	xxxxx	xxxxx	RBIT — 64-bit	-
1	0	00000	000001	xxxxx	xxxxx	REV16 — 64-bit	-
1	0	00000	000010	xxxxx	xxxxx	REV32	-
1	0	00000	000011	xxxxx	xxxxx	REV — 64-bit	-
1	0	00000	000100	xxxxx	xxxxx	CLZ — 64-bit	-
1	0	00000	000101	xxxxx	xxxxx	CLS — 64-bit	-
1	0	00000	000110	xxxxx	xxxxx	CTZ — 64-bit	FEAT_CSSC
1	0	00000	000111	xxxxx	xxxxx	CNT — 64-bit	FEAT_CSSC
1	0	00000	001000	xxxxx	xxxxx	ABS — 64-bit	FEAT_CSSC
1	0	00001	000000	xxxxx	xxxxx	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA	FEAT_PAuth
1	0	00001	000001	xxxxx	xxxxx	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB	FEAT_PAuth
1	0	00001	000010	xxxxx	xxxxx	PACDA, PACDZA — PACDA	FEAT_PAuth
1	0	00001	000011	xxxxx	xxxxx	PACDB, PACDZB — PACDB	FEAT_PAuth

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	opcode2	opcode	Rn	Rd	Instruction Details	Feature
1	0	00001	000100	xxxxx	xxxxx	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA	FEAT_PAuth
1	0	00001	000101	xxxxx	xxxxx	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB	FEAT_PAuth
1	0	00001	000110	xxxxx	xxxxx	AUTDA, AUTDZA — AUTDA	FEAT_PAuth
1	0	00001	000111	xxxxx	xxxxx	AUTDB, AUTDZB — AUTDB	FEAT_PAuth
1	0	00001	001xxx	!= 11111	xxxxx	UNALLOCATED	-
1	0	00001	001000	11111	xxxxx	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA	FEAT_PAuth
1	0	00001	001001	11111	xxxxx	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB	FEAT_PAuth
1	0	00001	001010	11111	xxxxx	PACDA, PACDZA — PACDZA	FEAT_PAuth
1	0	00001	001011	11111	xxxxx	PACDB, PACDZB — PACDZB	FEAT_PAuth
1	0	00001	001100	11111	xxxxx	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIZA	FEAT_PAuth
1	0	00001	001101	11111	xxxxx	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIZB	FEAT_PAuth
1	0	00001	001110	11111	xxxxx	AUTDA, AUTDZA — AUTDZA	FEAT_PAuth
1	0	00001	001111	11111	xxxxx	AUTDB, AUTDZB — AUTDZB	FEAT_PAuth
1	0	00001	01000x	!= 11111	xxxxx	UNALLOCATED	-
1	0	00001	010000	11111	xxxxx	XPACD, XPACI, XPACLRI — XPACI	FEAT_PAuth
1	0	00001	010001	11111	xxxxx	XPACD, XPACI, XPACLRI — XPACD	FEAT_PAuth
1	0	00001	01001x	xxxxx	xxxxx	UNALLOCATED	-
1	0	00001	0101xx	xxxxx	xxxxx	UNALLOCATED	-
1	0	00001	011xxx	xxxxx	xxxxx	UNALLOCATED	-
1	0	00001	10xxxx	xxxxx	!= 11110	UNALLOCATED	-
1	0	00001	1000xx	!= 11111	11110	UNALLOCATED	-
1	0	00001	100000	11111	11110	PACNBIASPPC	FEAT_PAuth_LR
1	0	00001	100001	11111	11110	PACNBIBSPPC	FEAT_PAuth_LR
1	0	00001	100010	11111	11110	PACIA171615	FEAT_PAuth_LR
1	0	00001	100011	11111	11110	PACIB171615	FEAT_PAuth_LR
1	0	00001	100100	xxxxx	11110	AUTIASPPCR	FEAT_PAuth_LR
1	0	00001	100101	xxxxx	11110	AUTIBSPPCR	FEAT_PAuth_LR
1	0	00001	10011x	xxxxx	11110	UNALLOCATED	-
1	0	00001	101xxx	!= 11111	11110	UNALLOCATED	-
1	0	00001	101000	11111	11110	PACIASPPC	FEAT_PAuth_LR
1	0	00001	101001	11111	11110	PACIBSPPC	FEAT_PAuth_LR
1	0	00001	10101x	11111	11110	UNALLOCATED	-
1	0	00001	10110x	11111	11110	UNALLOCATED	-
1	0	00001	101110	11111	11110	AUTIA171615	FEAT_PAuth_LR
1	0	00001	101111	11111	11110	AUTIB171615	FEAT_PAuth_LR
1	0	00001	11xxxx	xxxxx	xxxxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Logical (shifted register)

The encodings in this section are decoded from Data Processing – Register.

31	30 29	28	27 24	23 22	21	20 16	15 10	9 5	4 0
sf	opc	0	1 0 1 0	shift	N	Rm	imm6	Rn	Rd

sf	opc	N	Instruction Details
0	00	0	AND (shifted register) — 32-bit
0	00	1	BIC (shifted register) — 32-bit
0	01	0	ORR (shifted register) — 32-bit
0	01	1	ORN (shifted register) — 32-bit
0	10	0	EOR (shifted register) — 32-bit
0	10	1	EON (shifted register) — 32-bit
0	11	0	ANDS (shifted register) — 32-bit
0	11	1	BICS (shifted register) — 32-bit
1	00	0	AND (shifted register) — 64-bit
1	00	1	BIC (shifted register) — 64-bit
1	01	0	ORR (shifted register) — 64-bit
1	01	1	ORN (shifted register) — 64-bit
1	10	0	EOR (shifted register) — 64-bit
1	10	1	EON (shifted register) — 64-bit
1	11	0	ANDS (shifted register) — 64-bit
1	11	1	BICS (shifted register) — 64-bit

Add/subtract (shifted register)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27 24	23 22	21	20 16	15 10	9 5	4 0
sf	op	S	0	1 0 1 1	shift	0	Rm	imm6	Rn	Rd

sf	op	S	Instruction Details
0	0	0	ADD (shifted register) — 32-bit
0	0	1	ADDS (shifted register) — 32-bit

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	Instruction Details
0	1	0	SUB (shifted register) — 32-bit
0	1	1	SUBS (shifted register) — 32-bit
1	0	0	ADD (shifted register) — 64-bit
1	0	1	ADDS (shifted register) — 64-bit
1	1	0	SUB (shifted register) — 64-bit
1	1	1	SUBS (shifted register) — 64-bit

Add/subtract (extended register)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	24	23	22	21	20	16	15	13	12	10	9	5	4	0
sf	op	S	0	1 0 1 1		opt		1	Rm		option		imm3		Rn		Rd	

sf	op	S	opt	Instruction Details
x	x	x	!=00	UNALLOCATED
0	0	0	00	ADD (extended register) — 32-bit
0	0	1	00	ADDS (extended register) — 32-bit
0	1	0	00	SUB (extended register) — 32-bit
0	1	1	00	SUBS (extended register) — 32-bit
1	0	0	00	ADD (extended register) — 64-bit
1	0	1	00	ADDS (extended register) — 64-bit
1	1	0	00	SUB (extended register) — 64-bit
1	1	1	00	SUBS (extended register) — 64-bit

Add/subtract (with carry)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	10	9	5	4	0
sf	op	S	1	1 0 1		0 0 0 0		Rm		0 0 0 0 0 0		Rn		Rd	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	Instruction Details
0	0	0	ADC — 32-bit
0	0	1	ADCS — 32-bit
0	1	0	SBC — 32-bit
0	1	1	SBCS — 32-bit
1	0	0	ADC — 64-bit
1	0	1	ADCS — 64-bit
1	1	0	SBC — 64-bit
1	1	1	SBCS — 64-bit

Add/subtract (checked pointer)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	13	12	10	9	5	4	0
sf	op	S	1	1	0 1	0	0 0 0	Rm		0	0 1	imm3		Rn		Rd	

sf	op	S	Instruction Details	Feature
0	x	x	UNALLOCATED	-
1	x	1	UNALLOCATED	-
1	0	0	ADDPT	FEAT_CPA
1	1	0	SUBPT	FEAT_CPA

Rotate right into flags

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	15	14	10	9	5	4	3	0
sf	op	S	1	1	0 1	0	0 0 0	imm6		0	0 0 0 1	Rn		o2	mask	

sf	op	S	o2	Instruction Details	Feature
0	x	x	x	UNALLOCATED	-
1	0	0	x	UNALLOCATED	-
1	0	1	0	RMIF	FEAT_FlagM
1	0	1	1	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	o2	Instruction Details	Feature
1	1	x	x	UNALLOCATED	-

Evaluate into flags

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	15	14	13	10	9	5	4	3	0
sf	op	S	1	1 0 1		0 0 0 0		opcode2		sz	0 0 1 0		Rn		o3	mask	

sf	op	S	opcode2	sz	o3	mask	Instruction Details	Feature
0	0	0	xxxxxx	x	x	xxxx	UNALLOCATED	-
0	0	1	000000	x	0	!= 1101	UNALLOCATED	-
0	0	1	000000	x	1	xxxx	UNALLOCATED	-
0	0	1	000000	0	0	1101	SETF8, SETF16 — SETF8	FEAT_FlagM
0	0	1	000000	1	0	1101	SETF8, SETF16 — SETF16	FEAT_FlagM
0	0	1	!= 000000	x	x	xxxx	UNALLOCATED	-
0	1	x	xxxxxx	x	x	xxxx	UNALLOCATED	-
1	x	x	xxxxxx	x	x	xxxx	UNALLOCATED	-

Conditional compare (register)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	12	11	10	9	5	4	3	0
sf	op	S	1	1 0 1		0 0 1 0		Rm		cond		0	o2	Rn		o3	nzcv	

sf	op	S	o2	o3	Instruction Details
x	x	0	x	x	UNALLOCATED
x	x	1	0	1	UNALLOCATED
x	x	1	1	x	UNALLOCATED
0	0	1	0	0	CCMN (register) — 32-bit
0	1	1	0	0	CCMP (register) — 32-bit
1	0	1	0	0	CCMN (register) — 64-bit
1	1	1	0	0	CCMP (register) — 64-bit

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Conditional compare (immediate)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	12	11	10	9	5	4	3	0
sf	op	S	1	1 0 1		0 0 1 0		imm5		cond		1	o2	Rn		o3	nzcv	

sf	op	S	o2	o3	Instruction Details
x	x	0	x	x	UNALLOCATED
x	x	1	0	1	UNALLOCATED
x	x	1	1	x	UNALLOCATED
0	0	1	0	0	CCMN (immediate) — 32-bit
0	1	1	0	0	CCMP (immediate) — 32-bit
1	0	1	0	0	CCMN (immediate) — 64-bit
1	1	1	0	0	CCMP (immediate) — 64-bit

Conditional select

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	21	20	16	15	12	11	10	9	5	4	0
sf	op	S	1	1 0 1 0 1 0 0		Rm		cond		op2		Rn		Rd	

sf	op	S	op2	Instruction Details
x	x	0	1x	UNALLOCATED
x	x	1	xx	UNALLOCATED
0	0	0	00	CSEL — 32-bit
0	0	0	01	CSINC — 32-bit
0	1	0	00	CSINV — 32-bit
0	1	0	01	CSNEG — 32-bit
1	0	0	00	CSEL — 64-bit
1	0	0	01	CSINC — 64-bit
1	1	0	00	CSINV — 64-bit
1	1	0	01	CSNEG — 64-bit

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Data-processing (3 source)

The encodings in this section are decoded from Data Processing – Register.

31	30 29	28	27 24	23 21	20 16	15	14 10	9 5	4 0
sf	op54	1	1 0 1 1	op31	Rm	o0	Ra	Rn	Rd

sf	op54	op31	o0	Instruction Details	Feature
x	00	100	x	UNALLOCATED	-
x	!=00	xxx	x	UNALLOCATED	-
0	00	x01	x	UNALLOCATED	-
0	00	x1x	x	UNALLOCATED	-
0	00	000	0	MADD — 32-bit	-
0	00	000	1	MSUB — 32-bit	-
1	00	x10	1	UNALLOCATED	-
1	00	000	0	MADD — 64-bit	-
1	00	000	1	MSUB — 64-bit	-
1	00	001	0	SMADDL	-
1	00	001	1	SMSUBL	-
1	00	010	0	SMULH	-
1	00	011	0	MADDPT	FEAT_CPA
1	00	011	1	MSUBPT	FEAT_CPA
1	00	101	0	UMADDL	-
1	00	101	1	UMSUBL	-
1	00	110	0	UMULH	-
1	00	111	x	UNALLOCATED	-

Data Processing -- Scalar Floating-Point and Advanced SIMD

Data Processing – Scalar Floating-Point and Advanced SIMD

The encodings in this section are decoded from A64 instruction set encoding.

31 28	27 25	24 23	22 19	18 10	9 0
op0	1 1 1	op1	op2	op3	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
00x0	0x	x101	00xxxxx10	UNALLOCATED	-
00x0	11	xxxx	xxxxxxxx1	UNALLOCATED	-
0100	0x	x101	00xxxxx10	Cryptographic AES	-
0101	0x	x0xx	xxx0xxx00	Cryptographic three-register SHA	-
0101	0x	x0xx	xxx0xxx10	UNALLOCATED	-
0101	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0101	0x	x101	00xxxxx10	Cryptographic two-register SHA	-
0111	0x	x0xx	xxxxxxxx0	UNALLOCATED	-
011x	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxxx1	Advanced SIMD scalar copy	-
01x1	01	00xx	xxx0xxxx1	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	Advanced SIMD scalar three same FP16	-
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	Advanced SIMD scalar two-register miscellaneous FP16	-
01x1	0x	x0xx	xxx1xxxx1	Advanced SIMD scalar three same extra	-
01x1	0x	x100	00xxxxx10	Advanced SIMD scalar two-register miscellaneous	-
01x1	0x	x110	00xxxxx10	Advanced SIMD scalar pairwise	-
01x1	0x	x1xx	01xxxxx10	UNALLOCATED	-
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	Advanced SIMD scalar three different	-
01x1	0x	x1xx	xxxxxxxx1	Advanced SIMD scalar three same	-
01x1	10	xxxx	xxxxxxxx1	Advanced SIMD scalar shift by immediate	-
01x1	1x	xxxx	xxxxxxxx0	Advanced SIMD scalar x indexed element	-
01xx	11	xxxx	xxxxxxxx1	UNALLOCATED	-
0x00	0x	x0xx	xxx0xxx00	Advanced SIMD table lookup	-
0x00	0x	x0xx	xxx0xxx10	Advanced SIMD permute	-
0x10	0x	x0xx	xxx0xxxx0	Advanced SIMD extract	-
0xx0	00	00xx	xxx0xxxx1	Advanced SIMD copy	-
0xx0	01	00xx	xxx0xxxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	Advanced SIMD three same (FP16)	-
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	Advanced SIMD two-register miscellaneous (FP16)	-
0xx0	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxxx1	Advanced SIMD three-register extension	-
0xx0	0x	x100	00xxxxx10	Advanced SIMD two-register miscellaneous	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
0xx0	0x	x110	00xxxxx10	Advanced SIMD across lanes	-
0xx0	0x	x1xx	01xxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	Advanced SIMD three different	-
0xx0	0x	x1xx	xxxxxxxx1	Advanced SIMD three same	-
0xx0	10	0000	xxxxxxxx1	Advanced SIMD modified immediate	-
0xx0	10	!= 0000	xxxxxxxx1	Advanced SIMD shift by immediate	-
0xx0	1x	xxxx	xxxxxxxx0	Advanced SIMD vector x indexed element	-
10x0	xx	xxxx	xxxxxxxxx	UNALLOCATED	-
1100	00	0xxx	xxx1xxxxx	UNALLOCATED	-
1100	00	10xx	xxx10xxxx	Cryptographic three-register, imm2	-
1100	00	10xx	xxx11xxxx	UNALLOCATED	-
1100	00	11xx	xxx1x00xx	Cryptographic three-register SHA 512	-
1100	00	11xx	xxx1x01xx	UNALLOCATED	-
1100	00	11xx	xxx1x1xxx	UNALLOCATED	-
1100	00	xxxx	xxx0xxxxx	Cryptographic four-register	-
1100	01	00xx	xxxxxxxxx	XAR	FEAT_SHA3
1100	01	1000	0000xxxxx	UNALLOCATED	-
1100	01	1000	0001000xx	Cryptographic two-register SHA 512	-
1100	01	1000	0001100xx	UNALLOCATED	-
1100	01	1000	0001x01xx	UNALLOCATED	-
1100	01	1000	0001x1xxx	UNALLOCATED	-
1100	01	1000	001xxxxxx	UNALLOCATED	-
1100	01	1000	01xxxxxxx	UNALLOCATED	-
1100	01	1000	1xxxxxxxx	UNALLOCATED	-
1100	01	1001	xxxxxxxxx	UNALLOCATED	-
1100	01	101x	xxxxxxxxx	UNALLOCATED	-
1100	01	x1xx	xxxxxxxxx	UNALLOCATED	-
1100	1x	xxxx	xxxxxxxxx	UNALLOCATED	-
1110	xx	xxxx	xxxxxxxxx	UNALLOCATED	-
11x1	xx	xxxx	xxxxxxxxx	UNALLOCATED	-
x0x1	0x	x0xx	xxxxxxxxx	Conversion between floating-point and fixed-point	-
x0x1	0x	x1xx	xxx000000	Conversion between floating-point and integer	-
x0x1	0x	x1xx	xxx100000	UNALLOCATED	-
x0x1	0x	x1xx	xxxx10000	Floating-point data-processing (1 source)	-
x0x1	0x	x1xx	xxxxx1000	Floating-point compare	-
x0x1	0x	x1xx	xxxxxx100	Floating-point immediate	-
x0x1	0x	x1xx	xxxxxxx01	Floating-point conditional compare	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
x0x1	0x	x1xx	xxxxxxx10	Floating-point data-processing (2 source)	-
x0x1	0x	x1xx	xxxxxxx11	Floating-point conditional select	-
x0x1	1x	xxxx	xxxxxxxxx	Floating-point data-processing (3 source)	-

Cryptographic AES

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31 28	27 24	23 22	21 17	16 12	11 10	9 5	4 0
0 1 0 0	1 1 1 0	size	1 0 1 0 0	opcode	1 0	Rn	Rd

size	opcode	Instruction Details	Feature
00	000xx	UNALLOCATED	-
00	00100	AESE	FEAT_AES
00	00101	AESD	FEAT_AES
00	00110	AESMC	FEAT_AES
00	00111	AESIMC	FEAT_AES
00	01xxx	UNALLOCATED	-
00	1xxxx	UNALLOCATED	-
!=00	xxxxx	UNALLOCATED	-

Cryptographic three-register SHA

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31 28	27 24	23 22	21	20 16	15	14 12	11 10	9 5	4 0
0 1 0 1	1 1 1 0	size	0	Rm	0	opcode	0 0	Rn	Rd

size	opcode	Instruction Details	Feature
00	000	SHA1C	FEAT_SHA1
00	001	SHA1P	FEAT_SHA1
00	010	SHA1M	FEAT_SHA1
00	011	SHA1SU0	FEAT_SHA1
00	100	SHA256H	FEAT_SHA256

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opcode	Instruction Details	Feature
00	101	SHA256H2	FEAT_SHA256
00	110	SHA256SU1	FEAT_SHA256
00	111	UNALLOCATED	-
!=00	xxx	UNALLOCATED	-

Cryptographic two-register SHA

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–24	23–22	21–17	16–12	11–10	9–5	4–0
0 1 0 1	1 1 1 0	size	1 0 1 0 0	opcode	1 0	Rn	Rd

size	opcode	Instruction Details	Feature
00	00000	SHA1H	FEAT_SHA1
00	00001	SHA1SU1	FEAT_SHA1
00	00010	SHA256SU0	FEAT_SHA256
00	00011	UNALLOCATED	-
00	001xx	UNALLOCATED	-
00	01xxx	UNALLOCATED	-
00	1xxxx	UNALLOCATED	-
!=00	xxxxx	UNALLOCATED	-

Advanced SIMD scalar copy

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–30	29	28–25	24–23	22–21	20–16	15	14–11	10	9–5	4–0
0 1	op	1 1 1 1	0 0	0 0	imm5	0	imm4	1	Rn	Rd

op	imm4	Instruction Details	Feature
0	0000	DUP (element)	FEAT_AdvSIMD
0	!=0000	UNALLOCATED	-
1	xxxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Advanced SIMD scalar three same FP16

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15	14	13			11	10	9				5	4				0
0	1	U	1	1	1	1	0	a	1	0	Rm					0	0	opcode				1	Rn					Rd				

U	a	opcode	Instruction Details	Feature
x	x	00x	UNALLOCATED	-
x	0	010	UNALLOCATED	-
0	x	101	UNALLOCATED	-
0	0	011	FMULX	FEAT_AdvSIMD && FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_AdvSIMD && FEAT_FP16
0	0	110	UNALLOCATED	-
0	0	111	FRECPS	FEAT_AdvSIMD && FEAT_FP16
0	1	x10	UNALLOCATED	-
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	111	FRSQRTS	FEAT_AdvSIMD && FEAT_FP16
1	x	011	UNALLOCATED	-
1	x	11x	UNALLOCATED	-
1	0	100	FCMGE (register)	FEAT_AdvSIMD && FEAT_FP16
1	0	101	FACGE	FEAT_AdvSIMD && FEAT_FP16
1	1	010	FABD	FEAT_AdvSIMD && FEAT_FP16
1	1	100	FCMGT (register)	FEAT_AdvSIMD && FEAT_FP16
1	1	101	FACGT	FEAT_AdvSIMD && FEAT_FP16

Advanced SIMD scalar two-register miscellaneous FP16

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22				19	18	17	16				12	11	10	9				5	4				0
0	1	U	1	1	1	1	0	a	1	1	1	1		0	0	opcode					1	0	Rn					Rd				

U	a	opcode	Instruction Details	Feature
x	x	x0xxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Feature
x	x	1100x	UNALLOCATED	-
x	0	01xxx	UNALLOCATED	-
x	1	010xx	UNALLOCATED	-
x	1	11100	UNALLOCATED	-
0	0	11010	FCVTNS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	0	1111x	UNALLOCATED	-
0	1	01100	FCMGT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01111	UNALLOCATED	-
0	1	11010	FCVTPS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	11101	FRECPE	FEAT_AdvSIMD && FEAT_FP16
0	1	11110	UNALLOCATED	-
0	1	11111	FRECPX	FEAT_AdvSIMD && FEAT_FP16
1	x	1111x	UNALLOCATED	-
1	0	11010	FCVTNU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	0111x	UNALLOCATED	-
1	1	11010	FCVTPU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	11101	FRSQRTE	FEAT_AdvSIMD && FEAT_FP16

Advanced SIMD scalar three same extra

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15	14			11	10	9				5	4				0
0	1	U	1	1	1	1	0	size		0	Rm				1	opcode				1	Rn					Rd				

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	opcode	Instruction Details	Feature
0	xxxx	UNALLOCATED	-
1	0000	SQRDMLAH (vector)	FEAT_RDM
1	0001	SQRDMLSH (vector)	FEAT_RDM
1	001x	UNALLOCATED	-
1	01xx	UNALLOCATED	-
1	1xxx	UNALLOCATED	-

Advanced SIMD scalar two-register miscellaneous

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21				17	16			12	11	10	9			5	4			0
0	1	U	1	1	1	1	0	size		1	0	0	0	0	opcode				1	0	Rn				Rd			

U	size	opcode	Instruction Details	Feature
x	xx	00xx0	UNALLOCATED	-
x	xx	00x01	UNALLOCATED	-
x	xx	1x000	UNALLOCATED	-
x	xx	10xx1	UNALLOCATED	-
x	xx	11001	UNALLOCATED	-
x	0x	01xxx	UNALLOCATED	-
x	0x	1111x	UNALLOCATED	-
x	1x	01111	UNALLOCATED	-
x	1x	11100	UNALLOCATED	-
x	10	010x1	UNALLOCATED	-
x	10	01000	UNALLOCATED	-
0	xx	00011	SUQADD	FEAT_AdvSIMD
0	xx	00111	SQABS	FEAT_AdvSIMD
0	xx	10010	UNALLOCATED	-
0	xx	10100	SQXTN, SQXTN2	FEAT_AdvSIMD
0	xx	10110	UNALLOCATED	-
0	0x	11010	FCVTNS (vector)	FEAT_AdvSIMD
0	0x	11011	FCVTMS (vector)	FEAT_AdvSIMD
0	0x	11100	FCVTAS (vector)	FEAT_AdvSIMD
0	0x	11101	SCVTF (vector, integer)	FEAT_AdvSIMD
0	1x	01100	FCMGT (zero)	FEAT_AdvSIMD
0	1x	01101	FCMEQ (zero)	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
0	1x	01110	FCMLT (zero)	FEAT_AdvSIMD
0	1x	11010	FCVTPS (vector)	FEAT_AdvSIMD
0	1x	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD
0	1x	11101	FRECPE	FEAT_AdvSIMD
0	1x	11110	UNALLOCATED	-
0	1x	11111	FRECPX	FEAT_AdvSIMD
0	10	01010	UNALLOCATED	-
0	11	01000	CMGT (zero)	FEAT_AdvSIMD
0	11	01001	CMEQ (zero)	FEAT_AdvSIMD
0	11	01010	CMLT (zero)	FEAT_AdvSIMD
0	11	01011	ABS	FEAT_AdvSIMD
1	xx	00011	USQADD	FEAT_AdvSIMD
1	xx	00111	SQNEG	FEAT_AdvSIMD
1	xx	10010	SQXTUN, SQXTUN2	FEAT_AdvSIMD
1	xx	10100	UQXTN, UQXTN2	FEAT_AdvSIMD
1	0x	11010	FCVTNU (vector)	FEAT_AdvSIMD
1	0x	11011	FCVTMU (vector)	FEAT_AdvSIMD
1	0x	11100	FCVTAU (vector)	FEAT_AdvSIMD
1	0x	11101	UCVTF (vector, integer)	FEAT_AdvSIMD
1	00	10110	UNALLOCATED	-
1	01	10110	FCVTXN, FCVTXN2	FEAT_AdvSIMD
1	1x	01x10	UNALLOCATED	-
1	1x	01100	FCMGE (zero)	FEAT_AdvSIMD
1	1x	01101	FCMLE (zero)	FEAT_AdvSIMD
1	1x	1x110	UNALLOCATED	-
1	1x	11010	FCVTPU (vector)	FEAT_AdvSIMD
1	1x	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD
1	1x	11101	FRSQRTE	FEAT_AdvSIMD
1	1x	11111	UNALLOCATED	-
1	11	01000	CMGE (zero)	FEAT_AdvSIMD
1	11	01001	CMLE (zero)	FEAT_AdvSIMD
1	11	01011	NEG (vector)	FEAT_AdvSIMD

Advanced SIMD scalar pairwise

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21				17	16			12	11	10	9			5	4			0
0	1	U	1	1	1	1	0	size		1	1	0	0	0	opcode				1	0	Rn				Rd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
x	xx	x0xxx	UNALLOCATED	-
x	xx	01110	UNALLOCATED	-
x	xx	111xx	UNALLOCATED	-
x	10	01101	UNALLOCATED	-
0	x1	011x1	UNALLOCATED	-
0	x1	01100	UNALLOCATED	-
0	00	01100	FMAXNMP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	01101	FADDP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	01111	FMAXP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	10	01100	FMINNMP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	10	01111	FMINP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	11	x100x	UNALLOCATED	-
0	11	x1010	UNALLOCATED	-
0	11	01011	UNALLOCATED	-
0	11	11011	ADDP (scalar)	FEAT_AdvSIMD
0	!=11	x10xx	UNALLOCATED	-
1	xx	x10xx	UNALLOCATED	-
1	0x	01100	FMAXNMP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	0x	01101	FADDP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	0x	01111	FMAXP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	1x	01100	FMINNMP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	1x	01111	FMINP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	11	01101	UNALLOCATED	-

Advanced SIMD scalar three different

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15			12	11	10	9			5	4			0
0	1	U	1	1	1	1	0	size		1	Rm				opcode				0	0	Rn				Rd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	opcode	Instruction Details	Feature
0	0xxx	UNALLOCATED	-
0	1xx0	UNALLOCATED	-
0	1001	SQDMLAL, SQDMLAL2 (vector)	FEAT_AdvSIMD
0	1011	SQDMLSL, SQDMLSL2 (vector)	FEAT_AdvSIMD
0	1101	SQDMULL, SQDMULL2 (vector)	FEAT_AdvSIMD
0	1111	UNALLOCATED	-
1	xxxx	UNALLOCATED	-

Advanced SIMD scalar three same

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20	16	15	11	10	9	5	4	0
0	1	U	1	1	1	1	0	size		1	Rm		opcode		1	Rn		Rd	

U	size	opcode	Instruction Details	Feature
x	xx	x0011	UNALLOCATED	-
x	xx	011x1	UNALLOCATED	-
x	xx	11001	UNALLOCATED	-
x	xx	11110	UNALLOCATED	-
x	x0	0x1x0	UNALLOCATED	-
x	x1	00100	UNALLOCATED	-
x	x1	011x0	UNALLOCATED	-
x	0x	xx0x0	UNALLOCATED	-
x	0x	10x01	UNALLOCATED	-
x	01	00110	UNALLOCATED	-
x	1x	10010	UNALLOCATED	-
x	1x	11000	UNALLOCATED	-
x	10	0x0x0	UNALLOCATED	-
x	10	10x0x	UNALLOCATED	-
x	11	00000	UNALLOCATED	-
x	11	00010	UNALLOCATED	-
x	11	10101	UNALLOCATED	-
x	!=10	10100	UNALLOCATED	-
x	!=11	00111	UNALLOCATED	-
0	xx	00001	SQADD	FEAT_AdvSIMD
0	xx	00101	SQSUB	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
0	xx	01001	SQSHL (register)	FEAT_AdvSIMD
0	xx	01011	SQRSHL	FEAT_AdvSIMD
0	xx	10110	SQDMULH (vector)	FEAT_AdvSIMD
0	xx	10111	UNALLOCATED	-
0	xx	11101	UNALLOCATED	-
0	0x	11011	FMULX	FEAT_AdvSIMD
0	0x	11100	FCMEQ (register)	FEAT_AdvSIMD
0	0x	11111	FRECPS	FEAT_AdvSIMD
0	1x	11010	UNALLOCATED	-
0	1x	11011	UNALLOCATED	-
0	1x	11100	UNALLOCATED	-
0	1x	11111	FRSQRTS	FEAT_AdvSIMD
0	11	00110	CMGT (register)	FEAT_AdvSIMD
0	11	00111	CMGE (register)	FEAT_AdvSIMD
0	11	01000	SSHL	FEAT_AdvSIMD
0	11	01010	SRSHL	FEAT_AdvSIMD
0	11	10000	ADD (vector)	FEAT_AdvSIMD
0	11	10001	CMTST	FEAT_AdvSIMD
1	xx	00001	UQADD	FEAT_AdvSIMD
1	xx	00101	UQSUB	FEAT_AdvSIMD
1	xx	01001	UQSHL (register)	FEAT_AdvSIMD
1	xx	01011	UQRSHL	FEAT_AdvSIMD
1	xx	1x111	UNALLOCATED	-
1	xx	10110	SQRDMULH (vector)	FEAT_AdvSIMD
1	xx	11011	UNALLOCATED	-
1	0x	11100	FCMGE (register)	FEAT_AdvSIMD
1	0x	11101	FACGE	FEAT_AdvSIMD
1	1x	11010	FABD	FEAT_AdvSIMD
1	1x	11100	FCMGT (register)	FEAT_AdvSIMD
1	1x	11101	FACGT	FEAT_AdvSIMD
1	11	00110	CMHI (register)	FEAT_AdvSIMD
1	11	00111	CMHS (register)	FEAT_AdvSIMD
1	11	01000	USHL	FEAT_AdvSIMD
1	11	01010	URSHL	FEAT_AdvSIMD
1	11	10000	SUB (vector)	FEAT_AdvSIMD
1	11	10001	CMEQ (register)	FEAT_AdvSIMD

Advanced SIMD scalar shift by immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29	28			25	24	23	22	19	18	16	15	11	10	9	5	4	0
0	1	U	1	1	1	1	1	0	immh		immb		opcode		1	Rn		Rd	

U	immh	opcode	Instruction Details	Feature
x	xxxx	0xxx1	UNALLOCATED	-
x	xxxx	101xx	UNALLOCATED	-
x	xxxx	11101	UNALLOCATED	-
x	0xxx	x10x0	UNALLOCATED	-
x	0xxx	00xx0	UNALLOCATED	-
x	0xxx	110x1	UNALLOCATED	-
x	0000	x1110	UNALLOCATED	-
x	0000	1001x	UNALLOCATED	-
x	0000	11100	UNALLOCATED	-
x	0000	11111	UNALLOCATED	-
x	1xxx	110xx	UNALLOCATED	-
x	!= 0000	11110	UNALLOCATED	-
0	xxxx	01100	UNALLOCATED	-
0	xxxx	1000x	UNALLOCATED	-
0	1xxx	00000	SSHR	FEAT_AdvSIMD
0	1xxx	00010	SSRA	FEAT_AdvSIMD
0	1xxx	00100	SRSHR	FEAT_AdvSIMD
0	1xxx	00110	SRSRA	FEAT_AdvSIMD
0	1xxx	01000	UNALLOCATED	-
0	1xxx	01010	SHL	FEAT_AdvSIMD
0	!= 0000	01110	SQSHL (immediate)	FEAT_AdvSIMD
0	!= 0000	10010	SQSHRN, SQSHRN2	FEAT_AdvSIMD
0	!= 0000	10011	SQRSHRN, SQRSHRN2	FEAT_AdvSIMD
0	!= 0000	11100	SCVTF (vector, fixed-point)	FEAT_AdvSIMD
0	!= 0000	11111	FCVTZS (vector, fixed-point)	FEAT_AdvSIMD
1	0000	01100	UNALLOCATED	-
1	0000	1000x	UNALLOCATED	-
1	1xxx	00000	USHR	FEAT_AdvSIMD
1	1xxx	00010	USRA	FEAT_AdvSIMD
1	1xxx	00100	URSHR	FEAT_AdvSIMD
1	1xxx	00110	URSRA	FEAT_AdvSIMD
1	1xxx	01000	SRI	FEAT_AdvSIMD
1	1xxx	01010	SLI	FEAT_AdvSIMD
1	!= 0000	01100	SQSHLU	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	immh	opcode	Instruction Details	Feature
1	!=0000	01110	UQSHL (immediate)	FEAT_AdvSIMD
1	!=0000	10000	SQSHRUN, SQSHRUN2	FEAT_AdvSIMD
1	!=0000	10001	SQRSHRUN, SQRSHRUN2	FEAT_AdvSIMD
1	!=0000	10010	UQSHRN, UQSHRN2	FEAT_AdvSIMD
1	!=0000	10011	UQRSHRN, UQRSHRN2	FEAT_AdvSIMD
1	!=0000	11100	UCVTF (vector, fixed-point)	FEAT_AdvSIMD
1	!=0000	11111	FCVTZU (vector, fixed-point)	FEAT_AdvSIMD

Advanced SIMD scalar x indexed element

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31 30	29	28 24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0 1	U	1 1 1 1 1	size	L	M	Rm	opcode	H	0	Rn	Rd

U	size	opcode	Instruction Details	Feature
x	01	1001	UNALLOCATED	-
0	xx	0xx0	UNALLOCATED	-
0	xx	0011	SQDMLAL, SQDMLAL2 (by element)	FEAT_AdvSIMD
0	xx	0111	SQDMLSL, SQDMLSL2 (by element)	FEAT_AdvSIMD
0	xx	10x0	UNALLOCATED	-
0	xx	1011	SQDMULL, SQDMULL2 (by element)	FEAT_AdvSIMD
0	xx	1100	SQDMULH (by element)	FEAT_AdvSIMD
0	xx	1101	SQRDMULH (by element)	FEAT_AdvSIMD
0	xx	111x	UNALLOCATED	-
0	00	0001	FMLA (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	0101	FMLS (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	1001	FMUL (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	01	0x01	UNALLOCATED	-
0	1x	0001	FMLA (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
0	1x	0101	FMLS (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
0	1x	1001	FMUL (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
1	xx	0xxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
1	xx	1xx0	UNALLOCATED	-
1	xx	1011	UNALLOCATED	-
1	xx	1101	SQRDMLAH (by element)	FEAT_RDM
1	xx	1111	SQRDMLSH (by element)	FEAT_RDM
1	00	1001	FMULX (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
1	1x	1001	FMULX (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD

Advanced SIMD table lookup

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29					24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	op2		0	Rm		0	len		op	0	0	Rn		Rd	

Q	op2	len	op	Instruction Details	Feature
x	00	00	0	TBL — Single register table	FEAT_AdvSIMD
x	00	00	1	TBX — Single register table	FEAT_AdvSIMD
x	00	01	0	TBL — Two register table	FEAT_AdvSIMD
x	00	01	1	TBX — Two register table	FEAT_AdvSIMD
x	00	10	0	TBL — Three register table	FEAT_AdvSIMD
x	00	10	1	TBX — Three register table	FEAT_AdvSIMD
x	00	11	0	TBL — Four register table	FEAT_AdvSIMD
x	00	11	1	TBX — Four register table	FEAT_AdvSIMD
0	!= 00	xx	x	UNALLOCATED	-
1	01	xx	1	LUTI4 — Halfword	FEAT_AdvSIMD && FEAT_LUT
1	01	x0	0	UNALLOCATED	-
1	01	x1	0	LUTI4 — Byte	FEAT_AdvSIMD && FEAT_LUT
1	10	xx	0	UNALLOCATED	-
1	10	xx	1	LUTI2 — Byte	FEAT_AdvSIMD && FEAT_LUT
1	11	xx	x	LUTI2 — Halfword	FEAT_AdvSIMD && FEAT_LUT

Advanced SIMD permute

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29					24	23	22	21	20				16	15	14			12	11	10	9				5	4				0
0	Q	0	0	1	1	1	0	size		0	Rm					0	opcode				1	0	Rn						Rd			

opcode	Instruction Details	Feature
x00	UNALLOCATED	-
001	UZP1	FEAT_AdvSIMD
010	TRN1	FEAT_AdvSIMD
011	ZIP1	FEAT_AdvSIMD
101	UZP2	FEAT_AdvSIMD
110	TRN2	FEAT_AdvSIMD
111	ZIP2	FEAT_AdvSIMD

Advanced SIMD extract

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29					24	23	22	21	20				16	15	14				11	10	9				5	4				0
0	Q	1	0	1	1	1	0	op2		0	Rm					0	imm4					0	Rn						Rd			

op2	Instruction Details	Feature
00	EXT	FEAT_AdvSIMD
!= 00	UNALLOCATED	-

Advanced SIMD copy

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28			25	24	23	22	21	20				16	15	14				11	10	9				5	4				0
0	Q	op	0	1	1	1	0	0	0	0	imm5					0	imm4					1	Rn						Rd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	op	imm5	imm4	Instruction Details	Feature
x	0	xxxxx	0000	DUP (element)	FEAT_AdvSIMD
x	0	xxxxx	0001	DUP (general)	FEAT_AdvSIMD
x	0	xxxxx	01x0	UNALLOCATED	-
x	0	xxxxx	1xxx	UNALLOCATED	-
0	0	xxxxx	001x	UNALLOCATED	-
0	0	xxxxx	0101	SMOV — 32-bit	FEAT_AdvSIMD
0	0	xxxxx	0111	UMOV — 32-bit	FEAT_AdvSIMD
0	1	xxxxx	xxxx	UNALLOCATED	-
1	0	xxxxx	0010	UNALLOCATED	-
1	0	xxxxx	0011	INS (general)	FEAT_AdvSIMD
1	0	xxxxx	0101	SMOV — 64-bit	FEAT_AdvSIMD
1	0	x0xxx	0111	UNALLOCATED	-
1	0	x1000	0111	UMOV — 64-bit	FEAT_AdvSIMD
1	0	x1001	0111	UNALLOCATED	-
1	0	x101x	0111	UNALLOCATED	-
1	0	x11xx	0111	UNALLOCATED	-
1	1	xxxxx	xxxx	INS (element)	FEAT_AdvSIMD

Advanced SIMD three same (FP16)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15	14	13			11	10	9			5	4			0
0	Q	U	0	1	1	1	0	a	1	0	Rm				0	0	opcode				1	Rn				Rd			

U	a	opcode	Instruction Details	Feature
0	0	000	FMAXNM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	001	FMLA (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	010	FADD (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	011	FMULX	FEAT_AdvSIMD && FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_AdvSIMD && FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	110	FMAX (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	111	FRECPS	FEAT_AdvSIMD && FEAT_FP16
0	1	000	FMINNM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	001	FMLS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	010	FSUB (vector)	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Feature
0	1	011	FAMAX	FEAT_AdvSIMD && FEAT_FAMINMAX
0	1	10x	UNALLOCATED	-
0	1	110	FMIN (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	111	FRSQRTS	FEAT_AdvSIMD && FEAT_FP16
1	x	001	UNALLOCATED	-
1	0	000	FMAXNMP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	010	FADDP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	011	FMUL (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	100	FCMGE (register)	FEAT_AdvSIMD && FEAT_FP16
1	0	101	FACGE	FEAT_AdvSIMD && FEAT_FP16
1	0	110	FMAXP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	111	FDIV (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	000	FMINNMP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	010	FABD	FEAT_AdvSIMD && FEAT_FP16
1	1	011	FAMIN	FEAT_AdvSIMD && FEAT_FAMINMAX
1	1	100	FCMGT (register)	FEAT_AdvSIMD && FEAT_FP16
1	1	101	FACGT	FEAT_AdvSIMD && FEAT_FP16
1	1	110	FMINP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	111	FSCALE	FEAT_FP8

Advanced SIMD two-register miscellaneous (FP16)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22			19	18	17	16	12	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	a	1	1	1	1	0	0	opcode		1	0	Rn		Rd	

U	a	opcode	Instruction Details	Feature
x	x	x0xxx	UNALLOCATED	-
x	0	01xxx	UNALLOCATED	-
x	0	1111x	UNALLOCATED	-
x	1	010xx	UNALLOCATED	-
x	1	11100	UNALLOCATED	-
0	0	11000	FRINTN (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11001	FRINTM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11010	FCVTNS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Feature
0	0	11100	FCVTAS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01111	FABS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11000	FRINTP (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11001	FRINTZ (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	11101	FRECPE	FEAT_AdvSIMD && FEAT_FP16
0	1	1111x	UNALLOCATED	-
1	0	11000	FRINTA (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11001	FRINTX (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	x1110	UNALLOCATED	-
1	1	01100	FCMGE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01111	FNEG (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11000	UNALLOCATED	-
1	1	11001	FRINTI (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11010	FCVTPU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	11101	FRSQRTE	FEAT_AdvSIMD && FEAT_FP16
1	1	11111	FSQRT (vector)	FEAT_AdvSIMD && FEAT_FP16

Advanced SIMD three-register extension

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15	14			11	10	9			5	4			0
0	Q	U	0	1	1	1	0	size		0	Rm				1	opcode				1	Rn				Rd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Feature
x	0	xx	0000	UNALLOCATED	-
x	0	xx	0010	SDOT (vector)	FEAT_DotProd
x	0	0x	1010	UNALLOCATED	-
x	0	0x	1100	UNALLOCATED	-
x	0	00	1110	FCVTN, FCVTN2 (single-precision to 8-bit floating-point)	FEAT_FP8
x	0	00	1111	FDOT (8-bit floating-point to single-precision, vector)	FEAT_FP8DOT4
x	0	01	1110	FCVTN (half-precision to 8-bit floating-point)	FEAT_FP8
x	0	01	1111	FDOT (8-bit floating-point to half-precision, vector)	FEAT_FP8DOT2
x	0	10	0001	UNALLOCATED	-
x	0	10	0011	USDOT (vector)	FEAT_I8MM
x	0	10	1xx0	UNALLOCATED	-
x	0	10	1x11	UNALLOCATED	-
x	0	10	1001	UNALLOCATED	-
x	0	11	1xx0	UNALLOCATED	-
x	0	!=10	x0x1	UNALLOCATED	-
x	1	xx	0000	SQRDMLAH (vector)	FEAT_RDM
x	1	xx	0001	SQRDMLSH (vector)	FEAT_RDM
x	1	xx	0010	UDOT (vector)	FEAT_DotProd
x	1	xx	0011	UNALLOCATED	-
x	1	xx	10xx	FCMLA	FEAT_FCMA
x	1	xx	11x0	FCADD	FEAT_FCMA
x	1	x0	1111	UNALLOCATED	-
x	1	01	1111	BFDOT (vector)	FEAT_BF16
x	1	11	1111	BFMLALB, BFMLALT (vector)	FEAT_BF16
0	x	xx	01xx	UNALLOCATED	-
0	x	xx	1101	UNALLOCATED	-
0	0	00	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector) — FMLALLBB	FEAT_FP8FMA
0	0	01	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector) — FMLALLBT	FEAT_FP8FMA
0	0	11	1111	FMLALB, FMLALT (vector) — FMLALB	FEAT_FP8FMA
1	x	10	011x	UNALLOCATED	-
1	x	!=10	01xx	UNALLOCATED	-
1	0	00	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector) — FMLALLTB	FEAT_FP8FMA
1	0	01	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector) — FMLALLTT	FEAT_FP8FMA
1	0	10	0100	SMMLA (vector)	FEAT_I8MM

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Feature
1	0	10	0101	USMMLA (vector)	FEAT_I8MM
1	0	11	1101	UNALLOCATED	-
1	0	11	1111	FMLALB, FMLALT (vector) — FMLALT	FEAT_FP8FMA
1	0	!= 11	1101	UNALLOCATED	-
1	1	00	1101	FMMLA (widening, 8-bit floating-point to half-precision)	FEAT_F8F16MM
1	1	01	1101	BFMMLA (widening)	FEAT_BF16
1	1	10	0100	UMMLA (vector)	FEAT_I8MM
1	1	10	0101	UNALLOCATED	-
1	1	10	1101	FMMLA (widening, 8-bit floating-point to single-precision)	FEAT_F8F32MM
1	1	11	1101	UNALLOCATED	-

Advanced SIMD two-register miscellaneous

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21				17	16			12	11	10	9			5	4			0
0	Q	U	0	1	1	1	0	size		1	0	0	0	0	opcode				1	0	Rn				Rd			

U	size	opcode	Instruction Details	Feature
x	xx	1000x	UNALLOCATED	-
x	xx	10101	UNALLOCATED	-
x	0x	011xx	UNALLOCATED	-
x	10	11110	UNALLOCATED	-
x	11	1x110	UNALLOCATED	-
0	xx	00000	REV64	FEAT_AdvSIMD
0	xx	00001	REV16 (vector)	FEAT_AdvSIMD
0	xx	00010	SADDLP	FEAT_AdvSIMD
0	xx	00011	SUQADD	FEAT_AdvSIMD
0	xx	00100	CLS (vector)	FEAT_AdvSIMD
0	xx	00101	CNT	FEAT_AdvSIMD
0	xx	00110	SADALP	FEAT_AdvSIMD
0	xx	00111	SQABS	FEAT_AdvSIMD
0	xx	01000	CMGT (zero)	FEAT_AdvSIMD
0	xx	01001	CMEQ (zero)	FEAT_AdvSIMD
0	xx	01010	CMLT (zero)	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
0	xx	01011	ABS	FEAT_AdvSIMD
0	xx	10010	XTN, XTN2	FEAT_AdvSIMD
0	xx	10011	UNALLOCATED	-
0	xx	10100	SQXTN, SQXTN2	FEAT_AdvSIMD
0	0x	10110	FCVTN, FCVTN2 (double to single-precision, single to half-precision)	FEAT_AdvSIMD
0	0x	10111	FCVTL, FCVTL2	FEAT_AdvSIMD
0	0x	11000	FRINTN (vector)	FEAT_AdvSIMD
0	0x	11001	FRINTM (vector)	FEAT_AdvSIMD
0	0x	11010	FCVTNS (vector)	FEAT_AdvSIMD
0	0x	11011	FCVTMS (vector)	FEAT_AdvSIMD
0	0x	11100	FCVTAS (vector)	FEAT_AdvSIMD
0	0x	11101	SCVTF (vector, integer)	FEAT_AdvSIMD
0	0x	11110	FRINT32Z (vector)	FEAT_FRINTTS
0	0x	11111	FRINT64Z (vector)	FEAT_FRINTTS
0	1x	01100	FCMGT (zero)	FEAT_AdvSIMD
0	1x	01101	FCMEQ (zero)	FEAT_AdvSIMD
0	1x	01110	FCMLT (zero)	FEAT_AdvSIMD
0	1x	01111	FABS (vector)	FEAT_AdvSIMD
0	1x	1x111	UNALLOCATED	-
0	1x	11000	FRINTP (vector)	FEAT_AdvSIMD
0	1x	11001	FRINTZ (vector)	FEAT_AdvSIMD
0	1x	11010	FCVTPS (vector)	FEAT_AdvSIMD
0	1x	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD
0	1x	11100	URECPE	FEAT_AdvSIMD
0	1x	11101	FRECPE	FEAT_AdvSIMD
0	10	10110	BFCVTN, BFCVTN2	FEAT_BF16
1	xx	00000	REV32 (vector)	FEAT_AdvSIMD
1	xx	00010	UADDLP	FEAT_AdvSIMD
1	xx	00011	USQADD	FEAT_AdvSIMD
1	xx	00100	CLZ (vector)	FEAT_AdvSIMD
1	xx	00110	UADALP	FEAT_AdvSIMD
1	xx	00111	SQNEG	FEAT_AdvSIMD
1	xx	01000	CMGE (zero)	FEAT_AdvSIMD
1	xx	01001	CMLE (zero)	FEAT_AdvSIMD
1	xx	01010	UNALLOCATED	-
1	xx	01011	NEG (vector)	FEAT_AdvSIMD
1	xx	10010	SQXTUN, SQXTUN2	FEAT_AdvSIMD
1	xx	10011	SHLL, SHLL2	FEAT_AdvSIMD
1	xx	10100	UQXTN, UQXTN2	FEAT_AdvSIMD
1	x0	10110	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
1	0x	00001	UNALLOCATED	-
1	0x	11000	FRINTA (vector)	FEAT_AdvSIMD
1	0x	11001	FRINTX (vector)	FEAT_AdvSIMD
1	0x	11010	FCVTNU (vector)	FEAT_AdvSIMD
1	0x	11011	FCVTMU (vector)	FEAT_AdvSIMD
1	0x	11100	FCVTAU (vector)	FEAT_AdvSIMD
1	0x	11101	UCVTF (vector, integer)	FEAT_AdvSIMD
1	0x	11110	FRINT32X (vector)	FEAT_FRINTTS
1	0x	11111	FRINT64X (vector)	FEAT_FRINTTS
1	00	00101	NOT	FEAT_AdvSIMD
1	00	10111	F1CVTL, F1CVTL2, F2CVTL, F2CVTL2 — F1CVTL{2}	FEAT_FP8
1	01	00101	RBIT (vector)	FEAT_AdvSIMD
1	01	10110	FCVTXN, FCVTXN2	FEAT_AdvSIMD
1	01	10111	F1CVTL, F1CVTL2, F2CVTL, F2CVTL2 — F2CVTL{2}	FEAT_FP8
1	1x	00x01	UNALLOCATED	-
1	1x	01100	FCMGE (zero)	FEAT_AdvSIMD
1	1x	01101	FCMLE (zero)	FEAT_AdvSIMD
1	1x	01110	UNALLOCATED	-
1	1x	01111	FNEG (vector)	FEAT_AdvSIMD
1	1x	11000	UNALLOCATED	-
1	1x	11001	FRINTI (vector)	FEAT_AdvSIMD
1	1x	11010	FCVTPU (vector)	FEAT_AdvSIMD
1	1x	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD
1	1x	11100	URSQRTE	FEAT_AdvSIMD
1	1x	11101	FRSQRTE	FEAT_AdvSIMD
1	1x	11111	FSQRT (vector)	FEAT_AdvSIMD
1	10	10111	BF1CVTL, BF1CVTL2, BF2CVTL, BF2CVTL2 — BF1CVTL{2}	FEAT_FP8
1	11	10111	BF1CVTL, BF1CVTL2, BF2CVTL, BF2CVTL2 — BF2CVTL{2}	FEAT_FP8

Advanced SIMD across lanes

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21				17	16			12	11	10	9			5	4			0
0	Q	U	0	1	1	1	0	size		1	1	0	0	0	opcode				1	0	Rn				Rd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Feature
x	x	xx	x100x	UNALLOCATED	-
x	x	xx	000x0	UNALLOCATED	-
x	x	xx	00001	UNALLOCATED	-
x	x	xx	01011	UNALLOCATED	-
x	x	xx	100xx	UNALLOCATED	-
x	x	x0	001xx	UNALLOCATED	-
x	x	x0	01101	UNALLOCATED	-
x	x	x0	101xx	UNALLOCATED	-
x	x	x0	111xx	UNALLOCATED	-
x	x	x1	xx1xx	UNALLOCATED	-
x	0	xx	00011	SADDLV	FEAT_AdvSIMD
x	0	xx	01010	SMAXV	FEAT_AdvSIMD
x	0	xx	11010	SMINV	FEAT_AdvSIMD
x	0	xx	11011	ADDV	FEAT_AdvSIMD
x	0	x0	01110	UNALLOCATED	-
x	0	00	01100	FMAXNMV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	00	01111	FMAXV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	10	01100	FMINNMV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	10	01111	FMINV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	1	xx	00011	UADDLV	FEAT_AdvSIMD
x	1	xx	01010	UMAXV	FEAT_AdvSIMD
x	1	xx	11010	UMINV	FEAT_AdvSIMD
x	1	xx	11011	UNALLOCATED	-
0	1	x0	011x0	UNALLOCATED	-
0	1	x0	01111	UNALLOCATED	-
1	1	x0	01110	UNALLOCATED	-
1	1	00	01100	FMAXNMV — Single-precision	FEAT_AdvSIMD
1	1	00	01111	FMAXV — Single-precision	FEAT_AdvSIMD
1	1	10	01100	FMINNMV — Single-precision	FEAT_AdvSIMD
1	1	10	01111	FMINV — Single-precision	FEAT_AdvSIMD

Advanced SIMD three different

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20	16	15	12	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	size		1	Rm		opcode		0	0	Rn		Rd	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	opcode	Instruction Details	Feature
0	0000	SADDL, SADDL2	FEAT_AdvSIMD
0	0001	SADDW, SADDW2	FEAT_AdvSIMD
0	0010	SSUBL, SSUBL2	FEAT_AdvSIMD
0	0011	SSUBW, SSUBW2	FEAT_AdvSIMD
0	0100	ADDHN, ADDHN2	FEAT_AdvSIMD
0	0101	SABAL, SABAL2	FEAT_AdvSIMD
0	0110	SUBHN, SUBHN2	FEAT_AdvSIMD
0	0111	SABDL, SABDL2	FEAT_AdvSIMD
0	1000	SMLAL, SMLAL2 (vector)	FEAT_AdvSIMD
0	1001	SQDMLAL, SQDMLAL2 (vector)	FEAT_AdvSIMD
0	1010	SMLSL, SMLSL2 (vector)	FEAT_AdvSIMD
0	1011	SQDMLSL, SQDMLSL2 (vector)	FEAT_AdvSIMD
0	1100	SMULL, SMULL2 (vector)	FEAT_AdvSIMD
0	1101	SQDMULL, SQDMULL2 (vector)	FEAT_AdvSIMD
0	1110	PMULL, PMULL2	FEAT_AdvSIMD
0	1111	UNALLOCATED	-
1	0000	UADDL, UADDL2	FEAT_AdvSIMD
1	0001	UADDW, UADDW2	FEAT_AdvSIMD
1	0010	USUBL, USUBL2	FEAT_AdvSIMD
1	0011	USUBW, USUBW2	FEAT_AdvSIMD
1	0100	RADDHN, RADDHN2	FEAT_AdvSIMD
1	0101	UABAL, UABAL2	FEAT_AdvSIMD
1	0110	RSUBHN, RSUBHN2	FEAT_AdvSIMD
1	0111	UABDL, UABDL2	FEAT_AdvSIMD
1	1xx1	UNALLOCATED	-
1	1000	UMLAL, UMLAL2 (vector)	FEAT_AdvSIMD
1	1010	UMLSL, UMLSL2 (vector)	FEAT_AdvSIMD
1	1100	UMULL, UMULL2 (vector)	FEAT_AdvSIMD
1	1110	UNALLOCATED	-

Advanced SIMD three same

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15			11	10	9			5	4			0
0	Q	U	0	1	1	1	0	size		1	Rm				opcode				1	Rn				Rd			

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
0	xx	00000	SHADD	FEAT_AdvSIMD
0	xx	00001	SQADD	FEAT_AdvSIMD
0	xx	00010	SRHADD	FEAT_AdvSIMD
0	xx	00100	SHSUB	FEAT_AdvSIMD
0	xx	00101	SQSUB	FEAT_AdvSIMD
0	xx	00110	CMGT (register)	FEAT_AdvSIMD
0	xx	00111	CMGE (register)	FEAT_AdvSIMD
0	xx	01000	SSHL	FEAT_AdvSIMD
0	xx	01001	SQSHL (register)	FEAT_AdvSIMD
0	xx	01010	SRSHL	FEAT_AdvSIMD
0	xx	01011	SQRSHL	FEAT_AdvSIMD
0	xx	01100	SMAX	FEAT_AdvSIMD
0	xx	01101	SMIN	FEAT_AdvSIMD
0	xx	01110	SABD	FEAT_AdvSIMD
0	xx	01111	SABA	FEAT_AdvSIMD
0	xx	10000	ADD (vector)	FEAT_AdvSIMD
0	xx	10001	CMTST	FEAT_AdvSIMD
0	xx	10010	MLA (vector)	FEAT_AdvSIMD
0	xx	10011	MUL (vector)	FEAT_AdvSIMD
0	xx	10100	SMAXP	FEAT_AdvSIMD
0	xx	10101	SMINP	FEAT_AdvSIMD
0	xx	10110	SQDMULH (vector)	FEAT_AdvSIMD
0	xx	10111	ADDP (vector)	FEAT_AdvSIMD
0	x1	11101	UNALLOCATED	-
0	0x	11000	FMAXNM (vector)	FEAT_AdvSIMD
0	0x	11001	FMLA (vector)	FEAT_AdvSIMD
0	0x	11010	FADD (vector)	FEAT_AdvSIMD
0	0x	11011	FMULX	FEAT_AdvSIMD
0	0x	11100	FCMEQ (register)	FEAT_AdvSIMD
0	0x	11110	FMAX (vector)	FEAT_AdvSIMD
0	0x	11111	FRECPS	FEAT_AdvSIMD
0	00	00011	AND (vector)	FEAT_AdvSIMD
0	00	11101	FMLAL, FMLAL2 (vector) — FMLAL	FEAT_FHM
0	01	00011	BIC (vector, register)	FEAT_AdvSIMD
0	1x	11000	FMINNM (vector)	FEAT_AdvSIMD
0	1x	11001	FMLS (vector)	FEAT_AdvSIMD
0	1x	11010	FSUB (vector)	FEAT_AdvSIMD
0	1x	11011	FAMAX	FEAT_AdvSIMD && FEAT_FAMINMAX

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
0	1x	11100	UNALLOCATED	-
0	1x	11110	FMIN (vector)	FEAT_AdvSIMD
0	1x	11111	FRSQRTS	FEAT_AdvSIMD
0	10	00011	ORR (vector, register)	FEAT_AdvSIMD
0	10	11101	FMLSL, FMLSL2 (vector) — FMLSL	FEAT_FHM
0	11	00011	ORN (vector)	FEAT_AdvSIMD
1	xx	00000	UHADD	FEAT_AdvSIMD
1	xx	00001	UQADD	FEAT_AdvSIMD
1	xx	00010	URHADD	FEAT_AdvSIMD
1	xx	00100	UHSUB	FEAT_AdvSIMD
1	xx	00101	UQSUB	FEAT_AdvSIMD
1	xx	00110	CMHI (register)	FEAT_AdvSIMD
1	xx	00111	CMHS (register)	FEAT_AdvSIMD
1	xx	01000	USHL	FEAT_AdvSIMD
1	xx	01001	UQSHL (register)	FEAT_AdvSIMD
1	xx	01010	URSHL	FEAT_AdvSIMD
1	xx	01011	UQRSHL	FEAT_AdvSIMD
1	xx	01100	UMAX	FEAT_AdvSIMD
1	xx	01101	UMIN	FEAT_AdvSIMD
1	xx	01110	UABD	FEAT_AdvSIMD
1	xx	01111	UABA	FEAT_AdvSIMD
1	xx	10000	SUB (vector)	FEAT_AdvSIMD
1	xx	10001	CMEQ (register)	FEAT_AdvSIMD
1	xx	10010	MLS (vector)	FEAT_AdvSIMD
1	xx	10011	PMUL	FEAT_AdvSIMD
1	xx	10100	UMAXP	FEAT_AdvSIMD
1	xx	10101	UMINP	FEAT_AdvSIMD
1	xx	10110	SQRDMULH (vector)	FEAT_AdvSIMD
1	xx	10111	UNALLOCATED	-
1	x1	11001	UNALLOCATED	-
1	0x	11000	FMAXNMP (vector)	FEAT_AdvSIMD
1	0x	11010	FADDP (vector)	FEAT_AdvSIMD
1	0x	11011	FMUL (vector)	FEAT_AdvSIMD
1	0x	11100	FCMGE (register)	FEAT_AdvSIMD
1	0x	11101	FACGE	FEAT_AdvSIMD
1	0x	11110	FMAXP (vector)	FEAT_AdvSIMD
1	0x	11111	FDIV (vector)	FEAT_AdvSIMD
1	00	00011	EOR (vector)	FEAT_AdvSIMD
1	00	11001	FMLAL, FMLAL2 (vector) — FMLAL2	FEAT_FHM
1	01	00011	BSL	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
1	1x	11000	FMINNMP (vector)	FEAT_AdvSIMD
1	1x	11010	FABD	FEAT_AdvSIMD
1	1x	11011	FAMIN	FEAT_AdvSIMD && FEAT_FAMINMAX
1	1x	11100	FCMGT (register)	FEAT_AdvSIMD
1	1x	11101	FACGT	FEAT_AdvSIMD
1	1x	11110	FMINP (vector)	FEAT_AdvSIMD
1	1x	11111	FSCALE	FEAT_FP8
1	10	00011	BIT	FEAT_AdvSIMD
1	10	11001	FMLSL, FMLSL2 (vector) — FMLSL2	FEAT_FHM
1	11	00011	BIF	FEAT_AdvSIMD

Advanced SIMD modified immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28			25	24	23	22			19	18	17	16	15	12	11	10	9	8	7	6	5	4	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode		o2	1	d	e	f	g	h	Rd	

Q	op	cmode	o2	Instruction Details	Feature
x	0	0xx0	0	MOVI — 32-bit shifted immediate	FEAT_AdvSIMD
x	0	0xx1	0	ORR (vector, immediate) — 32-bit	FEAT_AdvSIMD
x	0	10x0	0	MOVI — 16-bit shifted immediate	FEAT_AdvSIMD
x	0	10x1	0	ORR (vector, immediate) — 16-bit	FEAT_AdvSIMD
x	0	110x	0	MOVI — 32-bit shifting ones	FEAT_AdvSIMD
x	0	1110	0	MOVI — 8-bit	FEAT_AdvSIMD
x	0	1111	0	FMOV (vector, immediate) — Single-precision	FEAT_AdvSIMD
x	0	1111	1	FMOV (vector, immediate) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	!= 1111	1	UNALLOCATED	-
x	1	xxxx	1	UNALLOCATED	-
x	1	0xx0	0	MVNI — 32-bit shifted immediate	FEAT_AdvSIMD
x	1	0xx1	0	BIC (vector, immediate) — 32-bit	FEAT_AdvSIMD
x	1	10x0	0	MVNI — 16-bit shifted immediate	FEAT_AdvSIMD
x	1	10x1	0	BIC (vector, immediate) — 16-bit	FEAT_AdvSIMD
x	1	110x	0	MVNI — 32-bit shifting ones	FEAT_AdvSIMD
0	1	1110	0	MOVI — 64-bit scalar	FEAT_AdvSIMD
0	1	1111	0	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	op	cmode	o2	Instruction Details	Feature
1	1	1110	0	MOVI — 64-bit vector	FEAT_AdvSIMD
1	1	1111	0	FMOV (vector, immediate) — Double-precision	FEAT_AdvSIMD

Advanced SIMD shift by immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 25	24 23	22 19	18 16	15 11	10	9 5	4 0
0	Q	U	0 1 1 1	1 0	!= 0000 (immh)	immb	opcode	1	Rn	Rd

The following constraints also apply to this encoding: immh != ’0000’

U	immh	opcode	Instruction Details	Feature
x	!= 0000	0xxx1	UNALLOCATED	-
x	!= 0000	1x110	UNALLOCATED	-
x	!= 0000	101x1	UNALLOCATED	-
x	!= 0000	110xx	UNALLOCATED	-
x	!= 0000	11101	UNALLOCATED	-
0	!= 0000	00000	SSHR	FEAT_AdvSIMD
0	!= 0000	00010	SSRA	FEAT_AdvSIMD
0	!= 0000	00100	SRSHR	FEAT_AdvSIMD
0	!= 0000	00110	SRSRA	FEAT_AdvSIMD
0	!= 0000	01x00	UNALLOCATED	-
0	!= 0000	01010	SHL	FEAT_AdvSIMD
0	!= 0000	01110	SQSHL (immediate)	FEAT_AdvSIMD
0	!= 0000	10000	SHRN, SHRN2	FEAT_AdvSIMD
0	!= 0000	10001	RSHRN, RSHRN2	FEAT_AdvSIMD
0	!= 0000	10010	SQSHRN, SQSHRN2	FEAT_AdvSIMD
0	!= 0000	10011	SQRSHRN, SQRSHRN2	FEAT_AdvSIMD
0	!= 0000	10100	SSHLL, SSHLL2	FEAT_AdvSIMD
0	!= 0000	11100	SCVTF (vector, fixed-point)	FEAT_AdvSIMD
0	!= 0000	11111	FCVTZS (vector, fixed-point)	FEAT_AdvSIMD
1	!= 0000	00000	USHR	FEAT_AdvSIMD
1	!= 0000	00010	USRA	FEAT_AdvSIMD
1	!= 0000	00100	URSHR	FEAT_AdvSIMD
1	!= 0000	00110	URSRA	FEAT_AdvSIMD
1	!= 0000	01000	SRI	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	immh	opcode	Instruction Details	Feature
1	!= 0000	01010	SLI	FEAT_AdvSIMD
1	!= 0000	01100	SQSHLU	FEAT_AdvSIMD
1	!= 0000	01110	UQSHL (immediate)	FEAT_AdvSIMD
1	!= 0000	10000	SQSHRUN, SQSHRUN2	FEAT_AdvSIMD
1	!= 0000	10001	SQRSHRUN, SQRSHRUN2	FEAT_AdvSIMD
1	!= 0000	10010	UQSHRN, UQSHRN2	FEAT_AdvSIMD
1	!= 0000	10011	UQRSHRN, UQRSHRN2	FEAT_AdvSIMD
1	!= 0000	10100	USHLL, USHLL2	FEAT_AdvSIMD
1	!= 0000	11100	UCVTF (vector, fixed-point)	FEAT_AdvSIMD
1	!= 0000	11111	FCVTZU (vector, fixed-point)	FEAT_AdvSIMD

Advanced SIMD vector x indexed element

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20	19			16	15			12	11	10	9			5	4			0
0	Q	U	0	1	1	1	1	size		L	M	Rm				opcode				H	0	Rn				Rd			

Q	U	size	opcode	Instruction Details	Feature
x	x	01	1001	UNALLOCATED	-
x	0	xx	0010	SMLAL, SMLAL2 (by element)	FEAT_AdvSIMD
x	0	xx	0011	SQDMLAL, SQDMLAL2 (by element)	FEAT_AdvSIMD
x	0	xx	0110	SMLSL, SMLSL2 (by element)	FEAT_AdvSIMD
x	0	xx	0111	SQDMLSL, SQDMLSL2 (by element)	FEAT_AdvSIMD
x	0	xx	1000	MUL (by element)	FEAT_AdvSIMD
x	0	xx	1010	SMULL, SMULL2 (by element)	FEAT_AdvSIMD
x	0	xx	1011	SQDMULL, SQDMULL2 (by element)	FEAT_AdvSIMD
x	0	xx	1100	SQDMULH (by element)	FEAT_AdvSIMD
x	0	xx	1101	SQRDMULH (by element)	FEAT_AdvSIMD
x	0	xx	1110	SDOT (by element)	FEAT_DotProd
x	0	00	0000	FDOT (8-bit floating-point to single-precision, by element)	FEAT_FP8DOT4
x	0	00	0001	FMLA (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	00	0101	FMLS (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	00	1001	FMUL (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Feature
x	0	00	1111	SUDOT (by element)	FEAT_I8MM
x	0	01	0x01	UNALLOCATED	-
x	0	01	0000	FDOT (8-bit floating-point to half-precision, by element)	FEAT_FP8DOT2
x	0	01	1111	BFDOT (by element)	FEAT_BF16
x	0	1x	0001	FMLA (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
x	0	1x	0101	FMLS (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
x	0	1x	1001	FMUL (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
x	0	10	0000	FMLAL, FMLAL2 (by element) — FMLAL	FEAT_FHM
x	0	10	0100	FMLSL, FMLSL2 (by element) — FMLSL	FEAT_FHM
x	0	10	1111	USDOT (by element)	FEAT_I8MM
x	0	11	1111	BFMLALB, BFMLALT (by element)	FEAT_BF16
x	0	!= 10	0100	UNALLOCATED	-
x	1	xx	0xx1	FCMLA (by element)	FEAT_FCMA
x	1	xx	0000	MLA (by element)	FEAT_AdvSIMD
x	1	xx	0010	UMLAL, UMLAL2 (by element)	FEAT_AdvSIMD
x	1	xx	0100	MLS (by element)	FEAT_AdvSIMD
x	1	xx	0110	UMLSL, UMLSL2 (by element)	FEAT_AdvSIMD
x	1	xx	1010	UMULL, UMULL2 (by element)	FEAT_AdvSIMD
x	1	xx	1011	UNALLOCATED	-
x	1	xx	1101	SQRDMLAH (by element)	FEAT_RDM
x	1	xx	1110	UDOT (by element)	FEAT_DotProd
x	1	xx	1111	SQRDMLSH (by element)	FEAT_RDM
x	1	0x	1100	UNALLOCATED	-
x	1	00	1001	FMULX (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
x	1	1x	1001	FMULX (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
x	1	10	1000	FMLAL, FMLAL2 (by element) — FMLAL2	FEAT_FHM
x	1	10	1100	FMLSL, FMLSL2 (by element) — FMLSL2	FEAT_FHM
x	1	11	1x00	UNALLOCATED	-
0	0	11	0000	FMLALB, FMLALT (by element) — FMLALB	FEAT_FP8FMA
0	1	00	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element) — FMLALLBB	FEAT_FP8FMA
0	1	01	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element) — FMLALLBT	FEAT_FP8FMA

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Feature
1	0	11	0000	FMLALB, FMLALT (by element) — FMLALT	FEAT_FP8FMA
1	1	00	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element) — FMLALLTB	FEAT_FP8FMA
1	1	01	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element) — FMLALLTT	FEAT_FP8FMA

Cryptographic three-register, imm2

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–25	24–23	22–21	20–16	15–14	13–12	11–10	9–5	4–0
1100	111	00	10	Rm	10	imm2	opcode	Rn	Rd

opcode	Instruction Details	Feature
00	SM3TT1A	FEAT_SM3
01	SM3TT1B	FEAT_SM3
10	SM3TT2A	FEAT_SM3
11	SM3TT2B	FEAT_SM3

Cryptographic three-register SHA 512

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–25	24–23	22–21	20–16	15	14	13–12	11–10	9–5	4–0
1100	111	00	11	Rm	1	O	00	opcode	Rn	Rd

O	opcode	Instruction Details	Feature
0	00	SHA512H	FEAT_SHA512
0	01	SHA512H2	FEAT_SHA512
0	10	SHA512SU1	FEAT_SHA512
0	11	RAX1	FEAT_SHA3
1	00	SM3PARTW1	FEAT_SM3
1	01	SM3PARTW2	FEAT_SM3

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

O	opcode	Instruction Details	Feature
1	10	SM4EKEY	FEAT_SM4
1	11	UNALLOCATED	-

Cryptographic four-register

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–25	24–23	22–21	20–16	15	14–10	9–5	4–0
1100	111	00	Op0	Rm	0	Ra	Rn	Rd

Op0	Instruction Details	Feature
00	EOR3	FEAT_SHA3
01	BCAX	FEAT_SHA3
10	SM3SS1	FEAT_SM3
11	UNALLOCATED	-

Cryptographic two-register SHA 512

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–25	24–23	22–19	18–12	11–10	9–5	4–0
1100	111	01	1000	0001000	opcode	Rn	Rd

opcode	Instruction Details	Feature
00	SHA512SU0	FEAT_SHA512
01	SM4E	FEAT_SM4
1x	UNALLOCATED	-

Conversion between floating-point and fixed-point

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28–24	23–22	21	20–19	18–16	15–10	9–5	4–0
sf	0	S	11110	ftype	0	rmode	opcode	scale	Rn	Rd

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
x	0	xx	xx	1xx	UNALLOCATED	-
x	0	10	xx	0xx	UNALLOCATED	-
x	0	!=10	x1	01x	UNALLOCATED	-
x	0	!=10	0x	00x	UNALLOCATED	-
x	0	!=10	10	0xx	UNALLOCATED	-
x	1	xx	xx	xxx	UNALLOCATED	-
0	0	00	00	010	SCVTF (scalar, fixed-point) — 32-bit to single-precision	FEAT_FP
0	0	00	00	011	UCVTF (scalar, fixed-point) — 32-bit to single-precision	FEAT_FP
0	0	00	11	000	FCVTZS (scalar, fixed-point) — Single-precision to 32-bit	FEAT_FP
0	0	00	11	001	FCVTZU (scalar, fixed-point) — Single-precision to 32-bit	FEAT_FP
0	0	01	00	010	SCVTF (scalar, fixed-point) — 32-bit to double-precision	FEAT_FP
0	0	01	00	011	UCVTF (scalar, fixed-point) — 32-bit to double-precision	FEAT_FP
0	0	01	11	000	FCVTZS (scalar, fixed-point) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	001	FCVTZU (scalar, fixed-point) — Double-precision to 32-bit	FEAT_FP
0	0	11	00	010	SCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	11	000	FCVTZS (scalar, fixed-point) — Half-precision to 32-bit	FEAT_FP16
0	0	11	11	001	FCVTZU (scalar, fixed-point) — Half-precision to 32-bit	FEAT_FP16
1	0	00	00	010	SCVTF (scalar, fixed-point) — 64-bit to single-precision	FEAT_FP
1	0	00	00	011	UCVTF (scalar, fixed-point) — 64-bit to single-precision	FEAT_FP
1	0	00	11	000	FCVTZS (scalar, fixed-point) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	001	FCVTZU (scalar, fixed-point) — Single-precision to 64-bit	FEAT_FP
1	0	01	00	010	SCVTF (scalar, fixed-point) — 64-bit to double-precision	FEAT_FP
1	0	01	00	011	UCVTF (scalar, fixed-point) — 64-bit to double-precision	FEAT_FP
1	0	01	11	000	FCVTZS (scalar, fixed-point) — Double-precision to 64-bit	FEAT_FP

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
1	0	01	11	001	FCVTZU (scalar, fixed-point) — Double-precision to 64-bit	FEAT_FP
1	0	11	00	010	SCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	11	000	FCVTZS (scalar, fixed-point) — Half-precision to 64-bit	FEAT_FP16
1	0	11	11	001	FCVTZU (scalar, fixed-point) — Half-precision to 64-bit	FEAT_FP16

Conversion between floating-point and integer

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	S	1	1	1	1	0	ftype		1	rmode		opcode			0	0	0	0	0	0	Rn			Rd		

sf	S	ftype	rmode	opcode	Instruction Details	Feature
x	0	10	x0	xxx	UNALLOCATED	-
x	0	10	x1	0xx	UNALLOCATED	-
x	0	10	x1	10x	UNALLOCATED	-
x	0	11	x1	11x	UNALLOCATED	-
x	0	!=10	01	10x	UNALLOCATED	-
x	1	xx	xx	xxx	UNALLOCATED	-
0	0	x0	01	11x	UNALLOCATED	-
0	0	00	00	000	FCVTNS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	001	FCVTNU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	010	SCVTF (scalar, integer) — 32-bit to single-precision	FEAT_FP
0	0	00	00	011	UCVTF (scalar, integer) — 32-bit to single-precision	FEAT_FP
0	0	00	00	100	FCVTAS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	101	FCVTAU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	110	FMOV (general) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	111	FMOV (general) — 32-bit to single-precision	FEAT_FP

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
0	0	00	01	000	FCVTPS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	01	001	FCVTPU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	1x	1xx	UNALLOCATED	-
0	0	00	10	000	FCVTMS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	10	001	FCVTMU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	11	000	FCVTZS (scalar, integer) — Single-precision to 32-bit	FEAT_FP
0	0	00	11	001	FCVTZU (scalar, integer) — Single-precision to 32-bit	FEAT_FP
0	0	00	!= 00	01x	UNALLOCATED	-
0	0	01	0x	11x	UNALLOCATED	-
0	0	01	00	000	FCVTNS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	001	FCVTNU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	010	SCVTF (scalar, integer) — 32-bit to double-precision	FEAT_FP
0	0	01	00	011	UCVTF (scalar, integer) — 32-bit to double-precision	FEAT_FP
0	0	01	00	100	FCVTAS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	101	FCVTAU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	01	000	FCVTPS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	01	001	FCVTPU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	01	010	FCVTNS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	01	011	FCVTNU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	000	FCVTMS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	10	001	FCVTMU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	10	010	FCVTPS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	011	FCVTPU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	100	FCVTMS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	101	FCVTMU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	110	FCVTZS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
0	0	01	10	111	FCVTZU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	11	000	FCVTZS (scalar, integer) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	001	FCVTZU (scalar, integer) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	010	FCVTAS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	11	011	FCVTAU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	11	100	SCVTF (scalar SIMD&FP) — 32-bit to double-precision	FEAT_FPRCVT
0	0	01	11	101	UCVTF (scalar SIMD&FP) — 32-bit to double-precision	FEAT_FPRCVT
0	0	01	11	110	FJCVTZS	FEAT_JSCVT
0	0	01	11	111	UNALLOCATED	-
0	0	10	11	11x	UNALLOCATED	-
0	0	11	00	000	FCVTNS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	001	FCVTNU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	010	SCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	100	FCVTAS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	101	FCVTAU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	110	FMOV (general) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	111	FMOV (general) — 32-bit to half-precision	FEAT_FP16
0	0	11	01	000	FCVTPS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	01	001	FCVTPU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	01	010	FCVTNS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	01	011	FCVTNU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	000	FCVTMS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	10	001	FCVTMU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	10	010	FCVTPS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	011	FCVTPU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
0	0	11	10	100	FCVTMS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	101	FCVTMU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	110	FCVTZS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	111	FCVTZU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	11	000	FCVTZS (scalar, integer) — Half-precision to 32-bit	FEAT_FP16
0	0	11	11	001	FCVTZU (scalar, integer) — Half-precision to 32-bit	FEAT_FP16
0	0	11	11	010	FCVTAS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	11	011	FCVTAU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	11	100	SCVTF (scalar SIMD&FP) — 32-bit to half-precision	FEAT_FPRCVT
0	0	11	11	101	UCVTF (scalar SIMD&FP) — 32-bit to half-precision	FEAT_FPRCVT
1	0	x0	11	11x	UNALLOCATED	-
1	0	00	0x	11x	UNALLOCATED	-
1	0	00	00	000	FCVTNS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	001	FCVTNU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	010	SCVTF (scalar, integer) — 64-bit to single-precision	FEAT_FP
1	0	00	00	011	UCVTF (scalar, integer) — 64-bit to single-precision	FEAT_FP
1	0	00	00	100	FCVTAS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	101	FCVTAU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	01	000	FCVTPS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	01	001	FCVTPU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	01	010	FCVTNS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	01	011	FCVTNU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	000	FCVTMS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	10	001	FCVTMU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	10	010	FCVTPS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
1	0	00	10	011	FCVTPU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	100	FCVTMS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	101	FCVTMU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	110	FCVTZS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	111	FCVTZU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	11	000	FCVTZS (scalar, integer) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	001	FCVTZU (scalar, integer) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	010	FCVTAS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	11	011	FCVTAU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	11	100	SCVTF (scalar SIMD&FP) — 64-bit to single-precision	FEAT_FPRCVT
1	0	00	11	101	UCVTF (scalar SIMD&FP) — 64-bit to single-precision	FEAT_FPRCVT
1	0	01	00	000	FCVTNS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	001	FCVTNU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	010	SCVTF (scalar, integer) — 64-bit to double-precision	FEAT_FP
1	0	01	00	011	UCVTF (scalar, integer) — 64-bit to double-precision	FEAT_FP
1	0	01	00	100	FCVTAS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	101	FCVTAU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	110	FMOV (general) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	111	FMOV (general) — 64-bit to double-precision	FEAT_FP
1	0	01	01	x1x	UNALLOCATED	-
1	0	01	01	000	FCVTPS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	01	001	FCVTPU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	1x	01x	UNALLOCATED	-
1	0	01	1x	1xx	UNALLOCATED	-
1	0	01	10	000	FCVTMS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	10	001	FCVTMU (scalar) — Double-precision to 64-bit	FEAT_FP

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
1	0	01	11	000	FCVTZS (scalar, integer) — Double-precision to 64-bit	FEAT_FP
1	0	01	11	001	FCVTZU (scalar, integer) — Double-precision to 64-bit	FEAT_FP
1	0	10	01	110	FMOV (general) — Top half of 128-bit to 64-bit	FEAT_FP
1	0	10	01	111	FMOV (general) — 64-bit to top half of 128-bit	FEAT_FP
1	0	11	00	000	FCVTNS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	001	FCVTNU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	010	SCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	100	FCVTAS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	101	FCVTAU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	110	FMOV (general) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	111	FMOV (general) — 64-bit to half-precision	FEAT_FP16
1	0	11	01	000	FCVTPS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	01	001	FCVTPU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	01	010	FCVTNS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	01	011	FCVTNU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	000	FCVTMS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	10	001	FCVTMU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	10	010	FCVTPS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	011	FCVTPU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	100	FCVTMS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	101	FCVTMU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	110	FCVTZS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	111	FCVTZU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	11	000	FCVTZS (scalar, integer) — Half-precision to 64-bit	FEAT_FP16

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
1	0	11	11	001	FCVTZU (scalar, integer) — Half-precision to 64-bit	FEAT_FP16
1	0	11	11	010	FCVTAS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	11	011	FCVTAU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	11	100	SCVTF (scalar SIMD&FP) — 64-bit to half-precision	FEAT_FPRCVT
1	0	11	11	101	UCVTF (scalar SIMD&FP) — 64-bit to half-precision	FEAT_FPRCVT

Floating-point data-processing (1 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28	23 22	21	20 15	14 10	9 5	4 0
M	0	S	1 1 1 1 0	ftype	1	opcode	1 0 0 0 0	Rn	Rd

M	S	ftype	opcode	Instruction Details	Feature
0	0	xx	1xxxxx	UNALLOCATED	-
0	0	0x	0101xx	UNALLOCATED	-
0	0	0x	011xxx	UNALLOCATED	-
0	0	00	000000	FMOV (register) — Single-precision	FEAT_FP
0	0	00	000001	FABS (scalar) — Single-precision	FEAT_FP
0	0	00	000010	FNEG (scalar) — Single-precision	FEAT_FP
0	0	00	000011	FSQRT (scalar) — Single-precision	FEAT_FP
0	0	00	0001x0	UNALLOCATED	-
0	0	00	000101	FCVT — Single-precision to double-precision	FEAT_FP
0	0	00	000111	FCVT — Single-precision to half-precision	FEAT_FP
0	0	00	001000	FRINTN (scalar) — Single-precision	FEAT_FP
0	0	00	001001	FRINTP (scalar) — Single-precision	FEAT_FP
0	0	00	001010	FRINTM (scalar) — Single-precision	FEAT_FP
0	0	00	001011	FRINTZ (scalar) — Single-precision	FEAT_FP
0	0	00	001100	FRINTA (scalar) — Single-precision	FEAT_FP
0	0	00	001110	FRINTX (scalar) — Single-precision	FEAT_FP
0	0	00	001111	FRINTI (scalar) — Single-precision	FEAT_FP
0	0	00	010000	FRINT32Z (scalar) — Single-precision	FEAT_FRINTTS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	opcode	Instruction Details	Feature
0	0	00	010001	FRINT32X (scalar) — Single-precision	FEAT_FRINTTS
0	0	00	010010	FRINT64Z (scalar) — Single-precision	FEAT_FRINTTS
0	0	00	010011	FRINT64X (scalar) — Single-precision	FEAT_FRINTTS
0	0	01	000000	FMOV (register) — Double-precision	FEAT_FP
0	0	01	000001	FABS (scalar) — Double-precision	FEAT_FP
0	0	01	000010	FNEG (scalar) — Double-precision	FEAT_FP
0	0	01	000011	FSQRT (scalar) — Double-precision	FEAT_FP
0	0	01	000100	FCVT — Double-precision to single-precision	FEAT_FP
0	0	01	000101	UNALLOCATED	-
0	0	01	000110	BFCVT	FEAT_BF16
0	0	01	000111	FCVT — Double-precision to half-precision	FEAT_FP
0	0	01	001000	FRINTN (scalar) — Double-precision	FEAT_FP
0	0	01	001001	FRINTP (scalar) — Double-precision	FEAT_FP
0	0	01	001010	FRINTM (scalar) — Double-precision	FEAT_FP
0	0	01	001011	FRINTZ (scalar) — Double-precision	FEAT_FP
0	0	01	001100	FRINTA (scalar) — Double-precision	FEAT_FP
0	0	01	001110	FRINTX (scalar) — Double-precision	FEAT_FP
0	0	01	001111	FRINTI (scalar) — Double-precision	FEAT_FP
0	0	01	010000	FRINT32Z (scalar) — Double-precision	FEAT_FRINTTS
0	0	01	010001	FRINT32X (scalar) — Double-precision	FEAT_FRINTTS
0	0	01	010010	FRINT64Z (scalar) — Double-precision	FEAT_FRINTTS
0	0	01	010011	FRINT64X (scalar) — Double-precision	FEAT_FRINTTS
0	0	10	0xxxxx	UNALLOCATED	-
0	0	11	000000	FMOV (register) — Half-precision	FEAT_FP16
0	0	11	000001	FABS (scalar) — Half-precision	FEAT_FP16
0	0	11	000010	FNEG (scalar) — Half-precision	FEAT_FP16
0	0	11	000011	FSQRT (scalar) — Half-precision	FEAT_FP16
0	0	11	000100	FCVT — Half-precision to single-precision	FEAT_FP
0	0	11	000101	FCVT — Half-precision to double-precision	FEAT_FP
0	0	11	00011x	UNALLOCATED	-
0	0	11	001000	FRINTN (scalar) — Half-precision	FEAT_FP16
0	0	11	001001	FRINTP (scalar) — Half-precision	FEAT_FP16
0	0	11	001010	FRINTM (scalar) — Half-precision	FEAT_FP16
0	0	11	001011	FRINTZ (scalar) — Half-precision	FEAT_FP16
0	0	11	001100	FRINTA (scalar) — Half-precision	FEAT_FP16
0	0	11	001110	FRINTX (scalar) — Half-precision	FEAT_FP16
0	0	11	001111	FRINTI (scalar) — Half-precision	FEAT_FP16
0	0	11	01xxxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	opcode	Instruction Details	Feature
0	0	!=10	001101	UNALLOCATED	-
0	1	xx	xxxxxx	UNALLOCATED	-
1	x	xx	xxxxxx	UNALLOCATED	-

Floating-point compare

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15	14	13			10	9			5	4			0
M	0	S	1	1	1	1	0	ftype		1	Rm				op		1	0	0	0	Rn				opcode2			

M	S	ftype	op	opcode2	Instruction Details	Feature
0	0	xx	00	xx001	UNALLOCATED	-
0	0	xx	00	xx01x	UNALLOCATED	-
0	0	xx	00	xx1xx	UNALLOCATED	-
0	0	xx	!=00	xxxxx	UNALLOCATED	-
0	0	00	00	00000	FCMP — Single-precision	FEAT_FP
0	0	00	00	01000	FCMP — Single-precision, zero	FEAT_FP
0	0	00	00	10000	FCMPE — Single-precision	FEAT_FP
0	0	00	00	11000	FCMPE — Single-precision, zero	FEAT_FP
0	0	01	00	00000	FCMP — Double-precision	FEAT_FP
0	0	01	00	01000	FCMP — Double-precision, zero	FEAT_FP
0	0	01	00	10000	FCMPE — Double-precision	FEAT_FP
0	0	01	00	11000	FCMPE — Double-precision, zero	FEAT_FP
0	0	10	00	xx000	UNALLOCATED	-
0	0	11	00	00000	FCMP — Half-precision	FEAT_FP16
0	0	11	00	01000	FCMP — Half-precision, zero	FEAT_FP16
0	0	11	00	10000	FCMPE — Half-precision	FEAT_FP16
0	0	11	00	11000	FCMPE — Half-precision, zero	FEAT_FP16
0	1	xx	xx	xxxxx	UNALLOCATED	-
1	x	xx	xx	xxxxx	UNALLOCATED	-

Floating-point immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29	28				24	23	22	21	20				13	12		10	9				5	4				0
M	0	S	1	1	1	1	0	ftype		1	imm8					1	0	0	imm5					Rd				

M	S	ftype	imm5	Instruction Details	Feature
0	0	xx	!= 00000	UNALLOCATED	-
0	0	00	00000	FMOV (scalar, immediate) — Single-precision	FEAT_FP
0	0	01	00000	FMOV (scalar, immediate) — Double-precision	FEAT_FP
0	0	10	00000	UNALLOCATED	-
0	0	11	00000	FMOV (scalar, immediate) — Half-precision	FEAT_FP16
0	1	xx	xxxxx	UNALLOCATED	-
1	x	xx	xxxxx	UNALLOCATED	-

Floating-point conditional compare

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15			12	11	10	9				5	4	3			0
M	0	S	1	1	1	1	0	ftype		1	Rm					cond				0	1	Rn					op	nzcv			

M	S	ftype	op	Instruction Details	Feature
0	0	00	0	FCCMP — Single-precision	FEAT_FP
0	0	00	1	FCCMPE — Single-precision	FEAT_FP
0	0	01	0	FCCMP — Double-precision	FEAT_FP
0	0	01	1	FCCMPE — Double-precision	FEAT_FP
0	0	10	x	UNALLOCATED	-
0	0	11	0	FCCMP — Half-precision	FEAT_FP16
0	0	11	1	FCCMPE — Half-precision	FEAT_FP16
0	1	xx	x	UNALLOCATED	-
1	x	xx	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Floating-point data-processing (2 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15			12	11	10	9				5	4				0
M	0	S	1	1	1	1	0	ftype		1	Rm					opcode				1	0	Rn					Rd				

M	S	ftype	opcode	Instruction Details	Feature
0	0	00	0000	FMUL (scalar) — Single-precision	FEAT_FP
0	0	00	0001	FDIV (scalar) — Single-precision	FEAT_FP
0	0	00	0010	FADD (scalar) — Single-precision	FEAT_FP
0	0	00	0011	FSUB (scalar) — Single-precision	FEAT_FP
0	0	00	0100	FMAX (scalar) — Single-precision	FEAT_FP
0	0	00	0101	FMIN (scalar) — Single-precision	FEAT_FP
0	0	00	0110	FMAXNM (scalar) — Single-precision	FEAT_FP
0	0	00	0111	FMINNM (scalar) — Single-precision	FEAT_FP
0	0	00	1000	FNMUL (scalar) — Single-precision	FEAT_FP
0	0	01	0000	FMUL (scalar) — Double-precision	FEAT_FP
0	0	01	0001	FDIV (scalar) — Double-precision	FEAT_FP
0	0	01	0010	FADD (scalar) — Double-precision	FEAT_FP
0	0	01	0011	FSUB (scalar) — Double-precision	FEAT_FP
0	0	01	0100	FMAX (scalar) — Double-precision	FEAT_FP
0	0	01	0101	FMIN (scalar) — Double-precision	FEAT_FP
0	0	01	0110	FMAXNM (scalar) — Double-precision	FEAT_FP
0	0	01	0111	FMINNM (scalar) — Double-precision	FEAT_FP
0	0	01	1000	FNMUL (scalar) — Double-precision	FEAT_FP
0	0	10	xxxx	UNALLOCATED	-
0	0	11	0000	FMUL (scalar) — Half-precision	FEAT_FP16
0	0	11	0001	FDIV (scalar) — Half-precision	FEAT_FP16
0	0	11	0010	FADD (scalar) — Half-precision	FEAT_FP16
0	0	11	0011	FSUB (scalar) — Half-precision	FEAT_FP16
0	0	11	0100	FMAX (scalar) — Half-precision	FEAT_FP16
0	0	11	0101	FMIN (scalar) — Half-precision	FEAT_FP16
0	0	11	0110	FMAXNM (scalar) — Half-precision	FEAT_FP16
0	0	11	0111	FMINNM (scalar) — Half-precision	FEAT_FP16
0	0	11	1000	FNMUL (scalar) — Half-precision	FEAT_FP16
0	0	!=10	1001	UNALLOCATED	-
0	0	!=10	101x	UNALLOCATED	-
0	0	!=10	11xx	UNALLOCATED	-
0	1	xx	xxxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	opcode	Instruction Details	Feature
1	x	xx	xxxx	UNALLOCATED	-

Floating-point conditional select

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15			12	11	10	9				5	4				0
M	0	S	1	1	1	1	0	ftype		1	Rm					cond				1	1	Rn					Rd				

M	S	ftype	Instruction Details	Feature
0	0	00	FCSEL — Single-precision	FEAT_FP
0	0	01	FCSEL — Double-precision	FEAT_FP
0	0	10	UNALLOCATED	-
0	0	11	FCSEL — Half-precision	FEAT_FP16
0	1	xx	UNALLOCATED	-
1	x	xx	UNALLOCATED	-

Floating-point data-processing (3 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15	14			10	9				5	4				0
M	0	S	1	1	1	1	1	ftype		o1	Rm					o0	Ra				Rn					Rd				

M	S	ftype	o1	o0	Instruction Details	Feature
0	0	00	0	0	FMADD — Single-precision	FEAT_FP
0	0	00	0	1	FMSUB — Single-precision	FEAT_FP
0	0	00	1	0	FNMADD — Single-precision	FEAT_FP
0	0	00	1	1	FNMSUB — Single-precision	FEAT_FP
0	0	01	0	0	FMADD — Double-precision	FEAT_FP
0	0	01	0	1	FMSUB — Double-precision	FEAT_FP
0	0	01	1	0	FNMADD — Double-precision	FEAT_FP
0	0	01	1	1	FNMSUB — Double-precision	FEAT_FP
0	0	10	x	x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	o1	o0	Instruction Details	Feature
0	0	11	0	0	FMADD — Half-precision	FEAT_FP16
0	0	11	0	1	FMSUB — Half-precision	FEAT_FP16
0	0	11	1	0	FNMADD — Half-precision	FEAT_FP16
0	0	11	1	1	FNMSUB — Half-precision	FEAT_FP16
0	1	xx	x	x	UNALLOCATED	-
1	x	xx	x	x	UNALLOCATED	-

Loads and Stores

Loads and Stores

The encodings in this section are decoded from A64 instruction set encoding.

31–28	27	26	25	24–10	9–0
op0	1	op1	0	op2	

op0	op1	op2	Instruction details
0000	0	1xx1xxxxxxxxxxx	UNALLOCATED
0001	1	1xx1xxxxxxxxxxx	UNALLOCATED
0x00	0	00x1xxxxxxxxxxx	Compare and swap pair
0x00	0	10x0xxxxxxxxxxx	UNALLOCATED
0x00	0	11x0xxxxxxxxxxx	Compare and swap pair (unprivileged)
0x00	1	00x000000xxxxxx	Advanced SIMD load/store multiple structures
0x00	1	00x000001xxxxxx	UNALLOCATED
0x00	1	01x0xxxxxxxxxxx	Advanced SIMD load/store multiple structures (post-indexed)
0x00	1	0xx1xxxxxxxxxxx	UNALLOCATED
0x00	1	10x10001xxxxxxx	UNALLOCATED
0x00	1	10x1001xxxxxxxx	UNALLOCATED
0x00	1	10x101xxxxxxxxx	UNALLOCATED
0x00	1	10x11xxxxxxxxxx	UNALLOCATED
0x00	1	10xx0000xxxxxxx	Advanced SIMD load/store single structure
0x00	1	11xxxxxxxxxxxxx	Advanced SIMD load/store single structure (post-indexed)
0x00	1	x0x00001xxxxxxx	UNALLOCATED
0x00	1	x0x0001xxxxxxxx	UNALLOCATED
0x00	1	x0x001xxxxxxxxx	UNALLOCATED
0x00	1	x0x01xxxxxxxxxx	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
0x01	0	1xx0xxxxxxxxx11	UNALLOCATED
0x01	0	1xx1xxxxx000010	RCW compare and swap
0x01	0	1xx1xxxxx000011	RCW compare and swap pair
0x01	0	1xx1xxxxx00011x	UNALLOCATED
0x01	0	1xx1xxxxx001x1x	UNALLOCATED
0x01	0	1xx1xxxxx01xx1x	UNALLOCATED
0x01	0	1xx1xxxxx1xxx1x	UNALLOCATED
0x01	0	1xx1xxxxxxxxx00	128-bit atomic memory operations
0x01	0	1xx1xxxxxxxxx01	Atomic memory operations (unprivileged)
1001	0	1xx0xxxxxxxxx11	UNALLOCATED
1001	1	1xx1xxxxxxxxxxx	UNALLOCATED
100x	0	1xx1xxxxxxxxxxx	UNALLOCATED
1101	0	1000111110xxx11	GCS load/store
1101	0	1100111110xxx11	UNALLOCATED
1101	0	1x000xxxxxxxx11	UNALLOCATED
1101	0	1x0010xxxxxxx11	UNALLOCATED
1101	0	1x00110xxxxxx11	UNALLOCATED
1101	0	1x001110xxxxx11	UNALLOCATED
1101	0	1x0011110xxxx11	UNALLOCATED
1101	0	1x00111111xxx11	UNALLOCATED
1101	0	1x10xxxxxxxxx11	UNALLOCATED
1101	0	1xx1xxxxxxxxxxx	Load/store memory tags
1x00	0	00x1xxxxxxxxxxx	Load/store exclusive pair
1x00	0	10x0xxxxxxxxxxx	Load/store exclusive register (unprivileged)
1x00	0	11x0xxxxxxxxxxx	Compare and swap (unprivileged)
1x00	1	xxxxxxxxxxxxxxx	UNALLOCATED
x100	0	1xx1xxxxxxxxxxx	UNALLOCATED
x101	1	1xx1xxxxxxxxxxx	UNALLOCATED
xx00	0	00x0xxxxxxxxxxx	Load/store exclusive register
xx00	0	01x0xxxxxxxxxxx	Load/store ordered
xx00	0	01x1xxxxxxxxxxx	Compare and swap
xx01	0	10x0xxxxxxxxx10	Load/store ordered register pair
xx01	0	11x000000000010	Load/store ordered (writeback)
xx01	0	11x000000000110	UNALLOCATED
xx01	0	11x000000001x10	UNALLOCATED
xx01	0	11x00000001xx10	UNALLOCATED
xx01	0	11x0000001xxx10	UNALLOCATED
xx01	0	11x000001xxxx10	UNALLOCATED
xx01	0	11x00001xxxxx10	UNALLOCATED
xx01	0	11x0001xxxxxx10	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
xx01	0	11x001xxxxxxx10	UNALLOCATED
xx01	0	11x01xxxxxxxx10	UNALLOCATED
xx01	0	1xx0xxxxxxxx00	Load/store ordered (unscaled immediate)
xx01	1	1xx0xxxxxxxx00	UNALLOCATED
xx01	1	1xx0xxxxxxxx10	Load/store ordered (SIMD&FP)
xx01	1	1xx0xxxxxxxx11	UNALLOCATED
xx01	x	0xxxxxxxxxxxxx	Load register (literal)
xx01	x	1xx0xxxxxxxx01	Memory Copy and Memory Set
xx10	x	00xxxxxxxxxxxx	Load/store no-allocate pair (offset)
xx10	x	01xxxxxxxxxxxx	Load/store register pair (post-indexed)
xx10	x	10xxxxxxxxxxxx	Load/store register pair (offset)
xx10	x	11xxxxxxxxxxxx	Load/store register pair (pre-indexed)
xx11	x	0xx0xxxxxxxx00	Load/store register (unscaled immediate)
xx11	x	0xx0xxxxxxxx01	Load/store register (immediate post-indexed)
xx11	x	0xx0xxxxxxxx10	Load/store register (unprivileged)
xx11	x	0xx0xxxxxxxx11	Load/store register (immediate pre-indexed)
xx11	x	0xx1xxxxxxxx00	Atomic memory operations
xx11	x	0xx1xxxxxxxx10	Load/store register (register offset)
xx11	x	0xx1xxxxxxxxx1	Load/store register (pac)
xx11	x	1xxxxxxxxxxxxx	Load/store register (unsigned immediate)

Compare and swap pair

The encodings in this section are decoded from Loads and Stores.

31	30	29			27	26	25			23	22	21	20			16	15	14			10	9			5	4			0
0	sz	0	0	1		0	0	0	0		L	1	Rs				o0	Rt2				Rn				Rt			

sz	L	o0	Rt2	Instruction Details	Feature
x	x	x	!= 11111	UNALLOCATED	-
0	0	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASP	FEAT_LSE
0	0	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPL	FEAT_LSE
0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPA	FEAT_LSE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	L	o0	Rt2	Instruction Details	Feature
0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL	FEAT_LSE
1	0	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASP	FEAT_LSE
1	0	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPL	FEAT_LSE
1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPA	FEAT_LSE
1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL	FEAT_LSE

Compare and swap pair (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
0	sz	0	0 1	0	0	1 1	L	0	Rs		o0	Rt2		Rn		Rt	

sz	L	o0	Rt2	Instruction Details	Feature
0	x	x	xxxxx	UNALLOCATED	-
1	x	x	!= 11111	UNALLOCATED	-
1	0	0	11111	CASPT, CASPAT, CASPALT, CASPLT — CASPT	FEAT_LSUI
1	0	1	11111	CASPT, CASPAT, CASPALT, CASPLT — CASPLT	FEAT_LSUI
1	1	0	11111	CASPT, CASPAT, CASPALT, CASPLT — CASPAT	FEAT_LSUI
1	1	1	11111	CASPT, CASPAT, CASPALT, CASPLT — CASPALT	FEAT_LSUI

Advanced SIMD load/store multiple structures

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	16	15	12	11	10	9	5	4	0
0	Q	0	0 1	1	0	0 0	L	0	0 0 0 0 0	opcode		size		Rn		Rt	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	opcode	Instruction Details	Feature
x	x0x1	UNALLOCATED	-
x	0101	UNALLOCATED	-
x	11xx	UNALLOCATED	-
0	0000	ST4 (multiple structures)	FEAT_AdvSIMD
0	0010	ST1 (multiple structures) — Four registers	FEAT_AdvSIMD
0	0100	ST3 (multiple structures)	FEAT_AdvSIMD
0	0110	ST1 (multiple structures) — Three registers	FEAT_AdvSIMD
0	0111	ST1 (multiple structures) — One register	FEAT_AdvSIMD
0	1000	ST2 (multiple structures)	FEAT_AdvSIMD
0	1010	ST1 (multiple structures) — Two registers	FEAT_AdvSIMD
1	0000	LD4 (multiple structures)	FEAT_AdvSIMD
1	0010	LD1 (multiple structures) — Four registers	FEAT_AdvSIMD
1	0100	LD3 (multiple structures)	FEAT_AdvSIMD
1	0110	LD1 (multiple structures) — Three registers	FEAT_AdvSIMD
1	0111	LD1 (multiple structures) — One register	FEAT_AdvSIMD
1	1000	LD2 (multiple structures)	FEAT_AdvSIMD
1	1010	LD1 (multiple structures) — Two registers	FEAT_AdvSIMD

Advanced SIMD load/store multiple structures (post-indexed)

The encodings in this section are decoded from Loads and Stores.

31	30	29 27	26	25 23	22	21	20 16	15 12	11 10	9 5	4 0
0	Q	0 0 1	1	0 0 1	L	0	Rm	opcode	size	Rn	Rt

L	Rm	opcode	Instruction Details	Feature
x	xxxxx	x0x1	UNALLOCATED	-
x	xxxxx	0101	UNALLOCATED	-
x	xxxxx	11xx	UNALLOCATED	-
0	11111	0000	ST4 (multiple structures) — Immediate offset	FEAT_AdvSIMD
0	11111	0010	ST1 (multiple structures) — Four registers, immediate offset	FEAT_AdvSIMD
0	11111	0100	ST3 (multiple structures) — Immediate offset	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	Rm	opcode	Instruction Details	Feature
0	11111	0110	ST1 (multiple structures) — Three registers, immediate offset	FEAT_AdvSIMD
0	11111	0111	ST1 (multiple structures) — One register, immediate offset	FEAT_AdvSIMD
0	11111	1000	ST2 (multiple structures) — Immediate offset	FEAT_AdvSIMD
0	11111	1010	ST1 (multiple structures) — Two registers, immediate offset	FEAT_AdvSIMD
0	!= 11111	0000	ST4 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!= 11111	0010	ST1 (multiple structures) — Four registers, register offset	FEAT_AdvSIMD
0	!= 11111	0100	ST3 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!= 11111	0110	ST1 (multiple structures) — Three registers, register offset	FEAT_AdvSIMD
0	!= 11111	0111	ST1 (multiple structures) — One register, register offset	FEAT_AdvSIMD
0	!= 11111	1000	ST2 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!= 11111	1010	ST1 (multiple structures) — Two registers, register offset	FEAT_AdvSIMD
1	11111	0000	LD4 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	0010	LD1 (multiple structures) — Four registers, immediate offset	FEAT_AdvSIMD
1	11111	0100	LD3 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	0110	LD1 (multiple structures) — Three registers, immediate offset	FEAT_AdvSIMD
1	11111	0111	LD1 (multiple structures) — One register, immediate offset	FEAT_AdvSIMD
1	11111	1000	LD2 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	1010	LD1 (multiple structures) — Two registers, immediate offset	FEAT_AdvSIMD
1	!= 11111	0000	LD4 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!= 11111	0010	LD1 (multiple structures) — Four registers, register offset	FEAT_AdvSIMD
1	!= 11111	0100	LD3 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!= 11111	0110	LD1 (multiple structures) — Three registers, register offset	FEAT_AdvSIMD
1	!= 11111	0111	LD1 (multiple structures) — One register, register offset	FEAT_AdvSIMD
1	!= 11111	1000	LD2 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!= 11111	1010	LD1 (multiple structures) — Two registers, register offset	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Advanced SIMD load/store single structure

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	17	16	15	13	12	11	10	9	5	4	0
0	Q	0 0	1	1	0 1	0	L	R	0 0	0 0	o2	opcode		S	size		Rn		Rt	

L	R	o2	opcode	S	size	Instruction Details	Feature
x	x	0	01x	x	x1	UNALLOCATED	-
x	x	0	10x	x	1x	UNALLOCATED	-
x	x	0	10x	1	01	UNALLOCATED	-
x	0	1	100	x	!=01	UNALLOCATED	-
x	0	1	100	1	01	UNALLOCATED	-
x	0	1	!=100	x	xx	UNALLOCATED	-
x	1	1	xxx	x	xx	UNALLOCATED	-
0	x	0	11x	x	xx	UNALLOCATED	-
0	0	0	000	x	xx	ST1 (single structure) — 8-bit	FEAT_AdvSIMD
0	0	0	001	x	xx	ST3 (single structure) — 8-bit	FEAT_AdvSIMD
0	0	0	010	x	x0	ST1 (single structure) — 16-bit	FEAT_AdvSIMD
0	0	0	011	x	x0	ST3 (single structure) — 16-bit	FEAT_AdvSIMD
0	0	0	100	x	00	ST1 (single structure) — 32-bit	FEAT_AdvSIMD
0	0	0	100	0	01	ST1 (single structure) — 64-bit	FEAT_AdvSIMD
0	0	0	101	x	00	ST3 (single structure) — 32-bit	FEAT_AdvSIMD
0	0	0	101	0	01	ST3 (single structure) — 64-bit	FEAT_AdvSIMD
0	0	1	100	0	01	STL1 (SIMD&FP)	FEAT_AdvSIMD && FEAT_LRCPC3
0	1	0	000	x	xx	ST2 (single structure) — 8-bit	FEAT_AdvSIMD
0	1	0	001	x	xx	ST4 (single structure) — 8-bit	FEAT_AdvSIMD
0	1	0	010	x	x0	ST2 (single structure) — 16-bit	FEAT_AdvSIMD
0	1	0	011	x	x0	ST4 (single structure) — 16-bit	FEAT_AdvSIMD
0	1	0	100	x	00	ST2 (single structure) — 32-bit	FEAT_AdvSIMD
0	1	0	100	0	01	ST2 (single structure) — 64-bit	FEAT_AdvSIMD
0	1	0	101	x	00	ST4 (single structure) — 32-bit	FEAT_AdvSIMD
0	1	0	101	0	01	ST4 (single structure) — 64-bit	FEAT_AdvSIMD
1	x	0	11x	1	xx	UNALLOCATED	-
1	0	0	000	x	xx	LD1 (single structure) — 8-bit	FEAT_AdvSIMD
1	0	0	001	x	xx	LD3 (single structure) — 8-bit	FEAT_AdvSIMD
1	0	0	010	x	x0	LD1 (single structure) — 16-bit	FEAT_AdvSIMD
1	0	0	011	x	x0	LD3 (single structure) — 16-bit	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	o2	opcode	S	size	Instruction Details	Feature
1	0	0	100	x	00	LD1 (single structure) — 32-bit	FEAT_AdvSIMD
1	0	0	100	0	01	LD1 (single structure) — 64-bit	FEAT_AdvSIMD
1	0	0	101	x	00	LD3 (single structure) — 32-bit	FEAT_AdvSIMD
1	0	0	101	0	01	LD3 (single structure) — 64-bit	FEAT_AdvSIMD
1	0	0	110	0	xx	LD1R	FEAT_AdvSIMD
1	0	0	111	0	xx	LD3R	FEAT_AdvSIMD
1	0	1	100	0	01	LDAP1 (SIMD&FP)	FEAT_AdvSIMD && FEAT_LRCPC3
1	1	0	000	x	xx	LD2 (single structure) — 8-bit	FEAT_AdvSIMD
1	1	0	001	x	xx	LD4 (single structure) — 8-bit	FEAT_AdvSIMD
1	1	0	010	x	x0	LD2 (single structure) — 16-bit	FEAT_AdvSIMD
1	1	0	011	x	x0	LD4 (single structure) — 16-bit	FEAT_AdvSIMD
1	1	0	100	x	00	LD2 (single structure) — 32-bit	FEAT_AdvSIMD
1	1	0	100	0	01	LD2 (single structure) — 64-bit	FEAT_AdvSIMD
1	1	0	101	x	00	LD4 (single structure) — 32-bit	FEAT_AdvSIMD
1	1	0	101	0	01	LD4 (single structure) — 64-bit	FEAT_AdvSIMD
1	1	0	110	0	xx	LD2R	FEAT_AdvSIMD
1	1	0	111	0	xx	LD4R	FEAT_AdvSIMD

Advanced SIMD load/store single structure (post-indexed)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	L	R	Rm		opcode		S	size	Rn		Rt

L	R	Rm	opcode	S	size	Instruction Details	Feature
x	x	xxxxx	01x	x	x1	UNALLOCATED	-
x	x	xxxxx	10x	x	1x	UNALLOCATED	-
x	x	xxxxx	10x	1	01	UNALLOCATED	-
0	x	xxxxx	11x	x	xx	UNALLOCATED	-
0	0	11111	000	x	xx	ST1 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	001	x	xx	ST3 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	010	x	x0	ST1 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	011	x	x0	ST3 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	Rm	opcode	S	size	Instruction Details	Feature
0	0	11111	100	x	00	ST1 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	100	0	01	ST1 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	101	x	00	ST3 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	101	0	01	ST3 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	0	!= 11111	000	x	xx	ST1 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	001	x	xx	ST3 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	010	x	x0	ST1 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	011	x	x0	ST3 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	100	x	00	ST1 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	100	0	01	ST1 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	101	x	00	ST3 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	101	0	01	ST3 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
0	1	11111	000	x	xx	ST2 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	001	x	xx	ST4 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	010	x	x0	ST2 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	011	x	x0	ST4 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	100	x	00	ST2 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	100	0	01	ST2 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	101	x	00	ST4 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	101	0	01	ST4 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	1	!= 11111	000	x	xx	ST2 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	001	x	xx	ST4 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	010	x	x0	ST2 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	011	x	x0	ST4 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	100	x	00	ST2 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	100	0	01	ST2 (single structure) — 64-bit, register offset	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	Rm	opcode	S	size	Instruction Details	Feature
0	1	!= 11111	101	x	00	ST4 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	101	0	01	ST4 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	x	xxxxx	11x	1	xx	UNALLOCATED	-
1	0	11111	000	x	xx	LD1 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	001	x	xx	LD3 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	010	x	x0	LD1 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	011	x	x0	LD3 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	100	x	00	LD1 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	100	0	01	LD1 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	101	x	00	LD3 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	101	0	01	LD3 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	110	0	xx	LD1R — Immediate offset	FEAT_AdvSIMD
1	0	11111	111	0	xx	LD3R — Immediate offset	FEAT_AdvSIMD
1	0	!= 11111	000	x	xx	LD1 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	001	x	xx	LD3 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	010	x	x0	LD1 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	011	x	x0	LD3 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	100	x	00	LD1 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	100	0	01	LD1 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	101	x	00	LD3 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	101	0	01	LD3 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	110	0	xx	LD1R — Register offset	FEAT_AdvSIMD
1	0	!= 11111	111	0	xx	LD3R — Register offset	FEAT_AdvSIMD
1	1	11111	000	x	xx	LD2 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	001	x	xx	LD4 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	010	x	x0	LD2 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	011	x	x0	LD4 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	100	x	00	LD2 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	Rm	opcode	S	size	Instruction Details	Feature
1	1	11111	100	0	01	LD2 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	101	x	00	LD4 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	101	0	01	LD4 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	110	0	xx	LD2R — Immediate offset	FEAT_AdvSIMD
1	1	11111	111	0	xx	LD4R — Immediate offset	FEAT_AdvSIMD
1	1	!= 11111	000	x	xx	LD2 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	001	x	xx	LD4 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	010	x	x0	LD2 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	011	x	x0	LD4 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	100	x	00	LD2 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	100	0	01	LD2 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	101	x	00	LD4 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	101	0	01	LD4 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	110	0	xx	LD2R — Register offset	FEAT_AdvSIMD
1	1	!= 11111	111	0	xx	LD4R — Register offset	FEAT_AdvSIMD

RCW compare and swap

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	24	23	22	21	20	16	15	10	9	5	4	0
0	S	0 1 1		0	0	1	A	R	1	Rs		0 0 0 0 1 0		Rn		Rt	

S	A	R	Instruction Details	Feature
0	0	0	RCWCAS, RCWCASA, RCWCASAL, RCWCASL — RCWCAS	FEAT_THE
0	0	1	RCWCAS, RCWCASA, RCWCASAL, RCWCASL — RCWCASL	FEAT_THE
0	1	0	RCWCAS, RCWCASA, RCWCASAL, RCWCASL — RCWCASA	FEAT_THE
0	1	1	RCWCAS, RCWCASA, RCWCASAL, RCWCASL — RCWCASAL	FEAT_THE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	A	R	Instruction Details	Feature
1	0	0	RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL — RCWSCAS	FEAT_THE
1	0	1	RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL — RCWSCASL	FEAT_THE
1	1	0	RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL — RCWSCASA	FEAT_THE
1	1	1	RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL — RCWSCASAL	FEAT_THE

RCW compare and swap pair

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	24	23	22	21	20	16	15	10	9	5	4	0
0	S	0	1	1	0	0	1	A	R	1	Rs	0	0	0	0	1	1	Rn	Rt

S	A	R	Instruction Details	Feature
0	0	0	RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL — RCWCASP	FEAT_D128 && FEAT_THE
0	0	1	RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL — RCWCASPL	FEAT_D128 && FEAT_THE
0	1	0	RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL — RCWCASPA	FEAT_D128 && FEAT_THE
0	1	1	RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL — RCWCASPAL	FEAT_D128 && FEAT_THE
1	0	0	RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL — RCWSCASP	FEAT_D128 && FEAT_THE
1	0	1	RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL — RCWSCASPL	FEAT_D128 && FEAT_THE
1	1	0	RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL — RCWSCASPA	FEAT_D128 && FEAT_THE
1	1	1	RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL — RCWSCASPAL	FEAT_D128 && FEAT_THE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

128-bit atomic memory operations

The encodings in this section are decoded from Loads and Stores.

31	30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0	S	0 1 1	0	0 1	A	R	1	Rt2	o3	opc	0 0	Rn	Rt

S	A	R	o3	opc	Instruction Details	Feature
x	x	x	x	1xx	UNALLOCATED	-
0	x	x	0	0x0	UNALLOCATED	-
0	0	0	0	001	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRP	FEAT_LSE128
0	0	0	0	011	LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETP	FEAT_LSE128
0	0	0	1	000	SWPP, SWPPA, SWPPAL, SWPPL — SWPP	FEAT_LSE128
0	0	0	1	001	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL — RCWCLRP	FEAT_D128 && FEAT_THE
0	0	0	1	010	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL — RCWSWPP	FEAT_D128 && FEAT_THE
0	0	0	1	011	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL — RCWSETP	FEAT_D128 && FEAT_THE
0	0	1	0	001	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPL	FEAT_LSE128
0	0	1	0	011	LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPL	FEAT_LSE128
0	0	1	1	000	SWPP, SWPPA, SWPPAL, SWPPL — SWPPL	FEAT_LSE128
0	0	1	1	001	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL — RCWCLRPL	FEAT_D128 && FEAT_THE
0	0	1	1	010	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL — RCWSWPPL	FEAT_D128 && FEAT_THE
0	0	1	1	011	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL — RCWSETPL	FEAT_D128 && FEAT_THE
0	1	0	0	001	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPA	FEAT_LSE128
0	1	0	0	011	LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPA	FEAT_LSE128
0	1	0	1	000	SWPP, SWPPA, SWPPAL, SWPPL — SWPPA	FEAT_LSE128
0	1	0	1	001	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL — RCWCLRPA	FEAT_D128 && FEAT_THE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	A	R	o3	opc	Instruction Details	Feature
0	1	0	1	010	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL — RCWSWPPA	FEAT_D128 && FEAT_THE
0	1	0	1	011	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL — RCWSETPA	FEAT_D128 && FEAT_THE
0	1	1	0	001	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPAL	FEAT_LSE128
0	1	1	0	011	LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPAL	FEAT_LSE128
0	1	1	1	000	SWPP, SWPPA, SWPPAL, SWPPL — SWPPAL	FEAT_LSE128
0	1	1	1	001	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL — RCWCLRPAL	FEAT_D128 && FEAT_THE
0	1	1	1	010	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL — RCWSWPPAL	FEAT_D128 && FEAT_THE
0	1	1	1	011	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL — RCWSETPAL	FEAT_D128 && FEAT_THE
1	x	x	0	0xx	UNALLOCATED	-
1	x	x	1	000	UNALLOCATED	-
1	0	0	1	001	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL — RCWSCLRP	FEAT_D128 && FEAT_THE
1	0	0	1	010	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL — RCWSSWPP	FEAT_D128 && FEAT_THE
1	0	0	1	011	RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL — RCWSSETP	FEAT_D128 && FEAT_THE
1	0	1	1	001	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL — RCWSCLRPL	FEAT_D128 && FEAT_THE
1	0	1	1	010	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL — RCWSSWPPL	FEAT_D128 && FEAT_THE
1	0	1	1	011	RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL — RCWSSETPL	FEAT_D128 && FEAT_THE
1	1	0	1	001	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL — RCWSCLRPA	FEAT_D128 && FEAT_THE
1	1	0	1	010	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL — RCWSSWPPA	FEAT_D128 && FEAT_THE
1	1	0	1	011	RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL — RCWSSETPA	FEAT_D128 && FEAT_THE
1	1	1	1	001	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL — RCWSCLRPAL	FEAT_D128 && FEAT_THE
1	1	1	1	010	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL — RCWSSWPPAL	FEAT_D128 && FEAT_THE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	A	R	o3	opc	Instruction Details	Feature
1	1	1	1	011	RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL — RCWSSETPAL	FEAT_D128 && FEAT_THE

Atomic memory operations (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	sz	0	1	1	0	0	1	A	R	1	Rs	o3	opc		0	1	Rn		Rt	

sz	A	R	o3	opc	Instruction Details	Feature
x	x	x	x	010	UNALLOCATED	-
x	x	x	x	1xx	UNALLOCATED	-
x	x	x	1	0x1	UNALLOCATED	-
0	0	0	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 32-bit no memory ordering	FEAT_LSUI
0	0	0	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 32-bit no memory ordering	FEAT_LSUI
0	0	0	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 32-bit no memory ordering	FEAT_LSUI
0	0	0	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 32-bit SWPT	FEAT_LSUI
0	0	1	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 32-bit release	FEAT_LSUI
0	0	1	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 32-bit release	FEAT_LSUI
0	0	1	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 32-bit release	FEAT_LSUI
0	0	1	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 32-bit SWPTL	FEAT_LSUI
0	1	0	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 32-bit acquire	FEAT_LSUI
0	1	0	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 32-bit acquire	FEAT_LSUI
0	1	0	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 32-bit acquire	FEAT_LSUI
0	1	0	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 32-bit SWPTA	FEAT_LSUI
0	1	1	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 32-bit acquire-release	FEAT_LSUI
0	1	1	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 32-bit acquire-release	FEAT_LSUI

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	A	R	o3	opc	Instruction Details	Feature
0	1	1	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 32-bit acquire-release	FEAT_LSUI
0	1	1	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 32-bit SWPTAL	FEAT_LSUI
1	0	0	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 64-bit no memory ordering	FEAT_LSUI
1	0	0	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 64-bit no memory ordering	FEAT_LSUI
1	0	0	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 64-bit no memory ordering	FEAT_LSUI
1	0	0	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 64-bit SWPT	FEAT_LSUI
1	0	1	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 64-bit release	FEAT_LSUI
1	0	1	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 64-bit release	FEAT_LSUI
1	0	1	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 64-bit release	FEAT_LSUI
1	0	1	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 64-bit SWPTL	FEAT_LSUI
1	1	0	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 64-bit acquire	FEAT_LSUI
1	1	0	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 64-bit acquire	FEAT_LSUI
1	1	0	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 64-bit acquire	FEAT_LSUI
1	1	0	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 64-bit SWPTA	FEAT_LSUI
1	1	1	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 64-bit acquire-release	FEAT_LSUI
1	1	1	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 64-bit acquire-release	FEAT_LSUI
1	1	1	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 64-bit acquire-release	FEAT_LSUI
1	1	1	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 64-bit SWPTAL	FEAT_LSUI

GCS load/store

The encodings in this section are decoded from Loads and Stores.

31			28	27	26	25									15	14		12	11	10	9		5	4		0
1	1	0	1	1	0	0	1	0	0	0	1	1	1	1	1	0	opc		1	1	Rn			Rt		

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Feature
000	GCSSTR	FEAT_GCS
001	GCSSTTR	FEAT_GCS
01x	UNALLOCATED	-
1xx	UNALLOCATED	-

Load/store memory tags

The encodings in this section are decoded from Loads and Stores.

31 28	27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
1 1 0 1	1	0	0 1	opc	1	imm9	op2	Rn	Rt

opc	imm9	op2	Instruction Details	Feature
00	xxxxxxxxx	01	STG — Post-index	FEAT_MTE
00	xxxxxxxxx	10	STG — Signed offset	FEAT_MTE
00	xxxxxxxxx	11	STG — Pre-index	FEAT_MTE
00	000000000	00	STZGM	FEAT_MTE2
01	xxxxxxxxx	00	LDG	FEAT_MTE
01	xxxxxxxxx	01	STZG — Post-index	FEAT_MTE
01	xxxxxxxxx	10	STZG — Signed offset	FEAT_MTE
01	xxxxxxxxx	11	STZG — Pre-index	FEAT_MTE
10	xxxxxxxxx	01	ST2G — Post-index	FEAT_MTE
10	xxxxxxxxx	10	ST2G — Signed offset	FEAT_MTE
10	xxxxxxxxx	11	ST2G — Pre-index	FEAT_MTE
10	000000000	00	STGM	FEAT_MTE2
11	xxxxxxxxx	01	STZ2G — Post-index	FEAT_MTE
11	xxxxxxxxx	10	STZ2G — Signed offset	FEAT_MTE
11	xxxxxxxxx	11	STZ2G — Pre-index	FEAT_MTE
11	000000000	00	LDGM	FEAT_MTE2
!= 01	!= 000000000	00	UNALLOCATED	-

Load/store exclusive pair

The encodings in this section are decoded from Loads and Stores.

31	30	29 27	26	25 23	22	21	20 16	15	14 10	9 5	4 0
1	sz	0 0 1	0	0 0 0	L	1	Rs	o0	Rt2	Rn	Rt

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	L	o0	Instruction Details
0	0	0	STXP — 32-bit
0	0	1	STLXP — 32-bit
0	1	0	LDXP — 32-bit
0	1	1	LDAXP — 32-bit
1	0	0	STXP — 64-bit
1	0	1	STLXP — 64-bit
1	1	0	LDXP — 64-bit
1	1	1	LDAXP — 64-bit

Load/store exclusive register (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
1	sz	0	0	1	0	0	1	0	L	0	Rs	o0	Rt2	Rn		Rt	

sz	L	o0	Instruction Details	Feature
0	0	0	STTXR — 32-bit	FEAT_LSUI
0	0	1	STLTXR — 32-bit	FEAT_LSUI
0	1	0	LDTXR — 32-bit	FEAT_LSUI
0	1	1	LDATXR — 32-bit	FEAT_LSUI
1	0	0	STTXR — 64-bit	FEAT_LSUI
1	0	1	STLTXR — 64-bit	FEAT_LSUI
1	1	0	LDTXR — 64-bit	FEAT_LSUI
1	1	1	LDATXR — 64-bit	FEAT_LSUI

Compare and swap (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
1	sz	0	0	1	0	0	1	1	L	0	Rs	o0	Rt2	Rn		Rt	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	L	o0	Rt2	Instruction Details	Feature
0	x	x	xxxxx	UNALLOCATED	-
1	x	x	!= 11111	UNALLOCATED	-
1	0	0	11111	CAST, CASAT, CASALT, CASLT — CAST	FEAT_LSUI
1	0	1	11111	CAST, CASAT, CASALT, CASLT — CASLT	FEAT_LSUI
1	1	0	11111	CAST, CASAT, CASALT, CASLT — CASAT	FEAT_LSUI
1	1	1	11111	CAST, CASAT, CASALT, CASLT — CASALT	FEAT_LSUI

Load/store exclusive register

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21	20 16	15	14 10	9 5	4 0
size	0 0 1	0	0 0 0	L	0	Rs	o0	Rt2	Rn	Rt

size	L	o0	Instruction Details
00	0	0	STXRB
00	0	1	STLXRB
00	1	0	LDXRB
00	1	1	LDAXRB
01	0	0	STXRH
01	0	1	STLXRH
01	1	0	LDXRH
01	1	1	LDAXRH
10	0	0	STXR — 32-bit
10	0	1	STLXR — 32-bit
10	1	0	LDXR — 32-bit
10	1	1	LDAXR — 32-bit
11	0	0	STXR — 64-bit
11	0	1	STLXR — 64-bit
11	1	0	LDXR — 64-bit
11	1	1	LDAXR — 64-bit

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Load/store ordered

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21	20 16	15	14 10	9 5	4 0
size	0 0 1	0	0 0 1	L	0	Rs	o0	Rt2	Rn	Rt

size	L	o0	Instruction Details	Feature
00	0	0	STLLRB	FEAT_LOR
00	0	1	STLRB	-
00	1	0	LDLARB	FEAT_LOR
00	1	1	LDARB	-
01	0	0	STLLRH	FEAT_LOR
01	0	1	STLRH	-
01	1	0	LDLARH	FEAT_LOR
01	1	1	LDARH	-
10	0	0	STLLR — 32-bit	FEAT_LOR
10	0	1	STLR — 32-bit	-
10	1	0	LDLAR — 32-bit	FEAT_LOR
10	1	1	LDAR — 32-bit	-
11	0	0	STLLR — 64-bit	FEAT_LOR
11	0	1	STLR — 64-bit	-
11	1	0	LDLAR — 64-bit	FEAT_LOR
11	1	1	LDAR — 64-bit	-

Compare and swap

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21	20 16	15	14 10	9 5	4 0
size	0 0 1	0	0 0 1	L	1	Rs	o0	Rt2	Rn	Rt

size	L	o0	Rt2	Instruction Details	Feature
xx	x	x	!= 11111	UNALLOCATED	-
00	0	0	11111	CASB, CASAB, CASALB, CASLB — CASB	FEAT_LSE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	L	o0	Rt2	Instruction Details	Feature
00	0	1	11111	CASB, CASAB, CASALB, CASLB — CASLB	FEAT_LSE
00	1	0	11111	CASB, CASAB, CASALB, CASLB — CASAB	FEAT_LSE
00	1	1	11111	CASB, CASAB, CASALB, CASLB — CASALB	FEAT_LSE
01	0	0	11111	CASH, CASAH, CASALH, CASLH — CASH	FEAT_LSE
01	0	1	11111	CASH, CASAH, CASALH, CASLH — CASLH	FEAT_LSE
01	1	0	11111	CASH, CASAH, CASALH, CASLH — CASAH	FEAT_LSE
01	1	1	11111	CASH, CASAH, CASALH, CASLH — CASALH	FEAT_LSE
10	0	0	11111	CAS, CASA, CASAL, CASL — 32-bit CAS	FEAT_LSE
10	0	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASL	FEAT_LSE
10	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit CASA	FEAT_LSE
10	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASAL	FEAT_LSE
11	0	0	11111	CAS, CASA, CASAL, CASL — 64-bit CAS	FEAT_LSE
11	0	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASL	FEAT_LSE
11	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit CASA	FEAT_LSE
11	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASAL	FEAT_LSE

Load/store ordered register pair

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21	20 16	15 12	11 10	9 5	4 0
size	0 1 1	0	0 1 0	L	0	Rt2	opc2	1 0	Rn	Rt

size	L	opc2	Instruction Details	Feature
0x	x	xxxx	UNALLOCATED	-
1x	x	001x	UNALLOCATED	-
1x	x	01xx	UNALLOCATED	-
1x	x	1xxx	UNALLOCATED	-
10	0	0000	STILP — 32-bit pre-index	FEAT_LRCPC3

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	L	opc2	Instruction Details	Feature
10	0	0001	STILP — 32-bit	FEAT_LRCPC3
10	1	0000	LDIAPP — 32-bit post-index	FEAT_LRCPC3
10	1	0001	LDIAPP — 32-bit	FEAT_LRCPC3
11	0	0000	STILP — 64-bit pre-index	FEAT_LRCPC3
11	0	0001	STILP — 64-bit	FEAT_LRCPC3
11	1	0000	LDIAPP — 64-bit post-index	FEAT_LRCPC3
11	1	0001	LDIAPP — 64-bit	FEAT_LRCPC3

Load/store ordered (writeback)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 10	9 5	4 0
size	0 1 1	0	0 1 1	L	0 0 0 0 0 0 0 0 0 0 1 0	Rn	Rt

size	L	Instruction Details	Feature
0x	x	UNALLOCATED	-
10	0	STLR — 32-bit	FEAT_LRCPC3
10	1	LDAPR — 32-bit	FEAT_LRCPC3
11	0	STLR — 64-bit	FEAT_LRCPC3
11	1	LDAPR — 64-bit	FEAT_LRCPC3

Load/store ordered (unscaled immediate)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	0 1 1	0	0 1	opc	0	imm9	0 0	Rn	Rt

size	opc	Instruction Details	Feature
00	00	STLURB	FEAT_LRCPC2
00	01	LDAPURB	FEAT_LRCPC2
00	10	LDAPURSB — 64-bit	FEAT_LRCPC2
00	11	LDAPURSB — 32-bit	FEAT_LRCPC2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Feature
01	00	STLURH	FEAT_LRCPC2
01	01	LDAPURH	FEAT_LRCPC2
01	10	LDAPURSH — 64-bit	FEAT_LRCPC2
01	11	LDAPURSH — 32-bit	FEAT_LRCPC2
10	00	STLUR — 32-bit	FEAT_LRCPC2
10	01	LDAPUR — 32-bit	FEAT_LRCPC2
10	10	LDAPURSW	FEAT_LRCPC2
10	11	UNALLOCATED	-
11	00	STLUR — 64-bit	FEAT_LRCPC2
11	01	LDAPUR — 64-bit	FEAT_LRCPC2
11	1x	UNALLOCATED	-

Load/store ordered (SIMD\&FP)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	0 1 1	1	0 1	opc	0	imm9	1 0	Rn	Rt

size	opc	Instruction Details	Feature
00	00	STLUR (SIMD&FP) — 8-bit	FEAT_FP && FEAT_LRCPC3
00	01	LDAPUR (SIMD&FP) — 8-bit	FEAT_FP && FEAT_LRCPC3
00	10	STLUR (SIMD&FP) — 128-bit	FEAT_FP && FEAT_LRCPC3
00	11	LDAPUR (SIMD&FP) — 128-bit	FEAT_FP && FEAT_LRCPC3
01	00	STLUR (SIMD&FP) — 16-bit	FEAT_FP && FEAT_LRCPC3
01	01	LDAPUR (SIMD&FP) — 16-bit	FEAT_FP && FEAT_LRCPC3
10	00	STLUR (SIMD&FP) — 32-bit	FEAT_FP && FEAT_LRCPC3
10	01	LDAPUR (SIMD&FP) — 32-bit	FEAT_FP && FEAT_LRCPC3
11	00	STLUR (SIMD&FP) — 64-bit	FEAT_FP && FEAT_LRCPC3
11	01	LDAPUR (SIMD&FP) — 64-bit	FEAT_FP && FEAT_LRCPC3
!= 00	1x	UNALLOCATED	-

Load register (literal)

The encodings in this section are decoded from Loads and Stores.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 30	29 27	26	25 24	23 5	4 0
opc	0 1 1	VR	0 0	imm19	Rt

opc	VR	Instruction Details	Feature
00	0	LDR (literal) — 32-bit	-
00	1	LDR (literal, SIMD&FP) — 32-bit	FEAT_FP
01	0	LDR (literal) — 64-bit	-
01	1	LDR (literal, SIMD&FP) — 64-bit	FEAT_FP
10	0	LDRSW (literal)	-
10	1	LDR (literal, SIMD&FP) — 128-bit	FEAT_FP
11	0	PRFM (literal)	-
11	1	UNALLOCATED	-

Memory Copy and Memory Set

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
size	0 1 1	o0	0 1	op1	0	Rs	op2	0 1	Rn	Rd

o0	op1	op2	Instruction Details	Feature
x	11	11xx	UNALLOCATED	-
0	00	0000	CPYFP, CPYFM, CPYFE — Prologue	FEAT_MOPS
0	00	0001	CPYFPWT, CPYFMWT, CPYFEWT — Prologue	FEAT_MOPS
0	00	0010	CPYFPRT, CPYFMRT, CPYFERT — Prologue	FEAT_MOPS
0	00	0011	CPYFPT, CPYFMT, CPYFET — Prologue	FEAT_MOPS
0	00	0100	CPYFPWN, CPYFMWN, CPYFEWN — Prologue	FEAT_MOPS
0	00	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — Prologue	FEAT_MOPS
0	00	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — Prologue	FEAT_MOPS
0	00	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — Prologue	FEAT_MOPS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Feature
0	00	1000	CPYFPRN, CPYFMRN, CPYFERN — Prologue	FEAT_MOPS
0	00	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — Prologue	FEAT_MOPS
0	00	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — Prologue	FEAT_MOPS
0	00	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — Prologue	FEAT_MOPS
0	00	1100	CPYFPN, CPYFMN, CPYFEN — Prologue	FEAT_MOPS
0	00	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — Prologue	FEAT_MOPS
0	00	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — Prologue	FEAT_MOPS
0	00	1111	CPYFPTN, CPYFMTN, CPYFETN — Prologue	FEAT_MOPS
0	01	0000	CPYFP, CPYFM, CPYFE — Main	FEAT_MOPS
0	01	0001	CPYFPWT, CPYFMWT, CPYFEWT — Main	FEAT_MOPS
0	01	0010	CPYFPRT, CPYFMRT, CPYFERT — Main	FEAT_MOPS
0	01	0011	CPYFPT, CPYFMT, CPYFET — Main	FEAT_MOPS
0	01	0100	CPYFPWN, CPYFMWN, CPYFEWN — Main	FEAT_MOPS
0	01	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — Main	FEAT_MOPS
0	01	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — Main	FEAT_MOPS
0	01	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — Main	FEAT_MOPS
0	01	1000	CPYFPRN, CPYFMRN, CPYFERN — Main	FEAT_MOPS
0	01	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — Main	FEAT_MOPS
0	01	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — Main	FEAT_MOPS
0	01	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — Main	FEAT_MOPS
0	01	1100	CPYFPN, CPYFMN, CPYFEN — Main	FEAT_MOPS
0	01	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — Main	FEAT_MOPS
0	01	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — Main	FEAT_MOPS
0	01	1111	CPYFPTN, CPYFMTN, CPYFETN — Main	FEAT_MOPS
0	10	0000	CPYFP, CPYFM, CPYFE — Epilogue	FEAT_MOPS
0	10	0001	CPYFPWT, CPYFMWT, CPYFEWT — Epilogue	FEAT_MOPS
0	10	0010	CPYFPRT, CPYFMRT, CPYFERT — Epilogue	FEAT_MOPS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Feature
0	10	0011	CPYFPT, CPYFMT, CPYFET — Epilogue	FEAT_MOPS
0	10	0100	CPYFPWN, CPYFMWN, CPYFEWN — Epilogue	FEAT_MOPS
0	10	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — Epilogue	FEAT_MOPS
0	10	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — Epilogue	FEAT_MOPS
0	10	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — Epilogue	FEAT_MOPS
0	10	1000	CPYFPRN, CPYFMRN, CPYFERN — Epilogue	FEAT_MOPS
0	10	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — Epilogue	FEAT_MOPS
0	10	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — Epilogue	FEAT_MOPS
0	10	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — Epilogue	FEAT_MOPS
0	10	1100	CPYFPN, CPYFMN, CPYFEN — Epilogue	FEAT_MOPS
0	10	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — Epilogue	FEAT_MOPS
0	10	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — Epilogue	FEAT_MOPS
0	10	1111	CPYFPTN, CPYFMTN, CPYFETN — Epilogue	FEAT_MOPS
0	11	0000	SETP, SETM, SETE — Prologue	FEAT_MOPS
0	11	0001	SETPT, SETMT, SETET — Prologue	FEAT_MOPS
0	11	0010	SETPN, SETMN, SETEN — Prologue	FEAT_MOPS
0	11	0011	SETPTN, SETMTN, SETETN — Prologue	FEAT_MOPS
0	11	0100	SETP, SETM, SETE — Main	FEAT_MOPS
0	11	0101	SETPT, SETMT, SETET — Main	FEAT_MOPS
0	11	0110	SETPN, SETMN, SETEN — Main	FEAT_MOPS
0	11	0111	SETPTN, SETMTN, SETETN — Main	FEAT_MOPS
0	11	1000	SETP, SETM, SETE — Epilogue	FEAT_MOPS
0	11	1001	SETPT, SETMT, SETET — Epilogue	FEAT_MOPS
0	11	1010	SETPN, SETMN, SETEN — Epilogue	FEAT_MOPS
0	11	1011	SETPTN, SETMTN, SETETN — Epilogue	FEAT_MOPS
1	00	0000	CPYP, CPYM, CPYE — Prologue	FEAT_MOPS
1	00	0001	CPYPWT, CPYMWT, CPYEWT — Prologue	FEAT_MOPS
1	00	0010	CPYPRT, CPYMRT, CPYERT — Prologue	FEAT_MOPS
1	00	0011	CPYPT, CPYMT, CPYET — Prologue	FEAT_MOPS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Feature
1	00	0100	CPYPWN, CPYMWN, CPYEWN — Prologue	FEAT_MOPS
1	00	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — Prologue	FEAT_MOPS
1	00	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — Prologue	FEAT_MOPS
1	00	0111	CPYPTWN, CPYMTWN, CPYETWN — Prologue	FEAT_MOPS
1	00	1000	CPYPRN, CPYMRN, CPYERN — Prologue	FEAT_MOPS
1	00	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — Prologue	FEAT_MOPS
1	00	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — Prologue	FEAT_MOPS
1	00	1011	CPYPTRN, CPYMTRN, CPYETRN — Prologue	FEAT_MOPS
1	00	1100	CPYPN, CPYMN, CPYEN — Prologue	FEAT_MOPS
1	00	1101	CPYPWTN, CPYMWTN, CPYEWTN — Prologue	FEAT_MOPS
1	00	1110	CPYPRTN, CPYMRTN, CPYERTN — Prologue	FEAT_MOPS
1	00	1111	CPYPTN, CPYMTN, CPYETN — Prologue	FEAT_MOPS
1	01	0000	CPYP, CPYM, CPYE — Main	FEAT_MOPS
1	01	0001	CPYPWT, CPYMWT, CPYEWT — Main	FEAT_MOPS
1	01	0010	CPYPRT, CPYMRT, CPYERT — Main	FEAT_MOPS
1	01	0011	CPYPT, CPYMT, CPYET — Main	FEAT_MOPS
1	01	0100	CPYPWN, CPYMWN, CPYEWN — Main	FEAT_MOPS
1	01	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — Main	FEAT_MOPS
1	01	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — Main	FEAT_MOPS
1	01	0111	CPYPTWN, CPYMTWN, CPYETWN — Main	FEAT_MOPS
1	01	1000	CPYPRN, CPYMRN, CPYERN — Main	FEAT_MOPS
1	01	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — Main	FEAT_MOPS
1	01	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — Main	FEAT_MOPS
1	01	1011	CPYPTRN, CPYMTRN, CPYETRN — Main	FEAT_MOPS
1	01	1100	CPYPN, CPYMN, CPYEN — Main	FEAT_MOPS
1	01	1101	CPYPWTN, CPYMWTN, CPYEWTN — Main	FEAT_MOPS
1	01	1110	CPYPRTN, CPYMRTN, CPYERTN — Main	FEAT_MOPS
1	01	1111	CPYPTN, CPYMTN, CPYETN — Main	FEAT_MOPS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Feature
1	10	0000	CPYP, CPYM, CPYE — Epilogue	FEAT_MOPS
1	10	0001	CPYPWT, CPYMWT, CPYEWT — Epilogue	FEAT_MOPS
1	10	0010	CPYPRT, CPYMRT, CPYERT — Epilogue	FEAT_MOPS
1	10	0011	CPYPT, CPYMT, CPYET — Epilogue	FEAT_MOPS
1	10	0100	CPYPWN, CPYMWN, CPYEWN — Epilogue	FEAT_MOPS
1	10	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — Epilogue	FEAT_MOPS
1	10	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — Epilogue	FEAT_MOPS
1	10	0111	CPYPTWN, CPYMTWN, CPYETWN — Epilogue	FEAT_MOPS
1	10	1000	CPYPRN, CPYMRN, CPYERN — Epilogue	FEAT_MOPS
1	10	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — Epilogue	FEAT_MOPS
1	10	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — Epilogue	FEAT_MOPS
1	10	1011	CPYPTRN, CPYMTRN, CPYETRN — Epilogue	FEAT_MOPS
1	10	1100	CPYPN, CPYMN, CPYEN — Epilogue	FEAT_MOPS
1	10	1101	CPYPWTN, CPYMWTN, CPYEWTN — Epilogue	FEAT_MOPS
1	10	1110	CPYPRTN, CPYMRTN, CPYERTN — Epilogue	FEAT_MOPS
1	10	1111	CPYPTN, CPYMTN, CPYETN — Epilogue	FEAT_MOPS
1	11	0000	SETGP, SETGM, SETGE — Prologue	FEAT_MOPS && FEAT_MTE
1	11	0001	SETGPT, SETGMT, SETGET — Prologue	FEAT_MOPS && FEAT_MTE
1	11	0010	SETGPN, SETGMN, SETGEN — Prologue	FEAT_MOPS && FEAT_MTE
1	11	0011	SETGPTN, SETGMTN, SETGETN — Prologue	FEAT_MOPS && FEAT_MTE
1	11	0100	SETGP, SETGM, SETGE — Main	FEAT_MOPS && FEAT_MTE
1	11	0101	SETGPT, SETGMT, SETGET — Main	FEAT_MOPS && FEAT_MTE
1	11	0110	SETGPN, SETGMN, SETGEN — Main	FEAT_MOPS && FEAT_MTE
1	11	0111	SETGPTN, SETGMTN, SETGETN — Main	FEAT_MOPS && FEAT_MTE
1	11	1000	SETGP, SETGM, SETGE — Epilogue	FEAT_MOPS && FEAT_MTE
1	11	1001	SETGPT, SETGMT, SETGET — Epilogue	FEAT_MOPS && FEAT_MTE
1	11	1010	SETGPN, SETGMN, SETGEN — Epilogue	FEAT_MOPS && FEAT_MTE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Feature
1	11	1011	SETGPTN, SETGMTN, SETGETN — Epilogue	FEAT_MOPS && FEAT_MTE

Load/store no-allocate pair (offset)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	VR	0 0 0	L	imm7	Rt2	Rn	Rt

opc	VR	L	Instruction Details	Feature
00	0	0	STNP — 32-bit	-
00	0	1	LDNP — 32-bit	-
00	1	0	STNP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDNP (SIMD&FP) — 32-bit	FEAT_FP
01	0	x	UNALLOCATED	-
01	1	0	STNP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDNP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STNP — 64-bit	-
10	0	1	LDNP — 64-bit	-
10	1	0	STNP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDNP (SIMD&FP) — 128-bit	FEAT_FP
11	0	0	STTNP	FEAT_LSUI
11	0	1	LDTNP	FEAT_LSUI
11	1	0	STTNP (SIMD&FP)	FEAT_FP && FEAT_LSUI
11	1	1	LDTNP (SIMD&FP)	FEAT_FP && FEAT_LSUI

Load/store register pair (post-indexed)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	VR	0 0 1	L	imm7	Rt2	Rn	Rt

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	VR	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP
11	0	0	STTP	FEAT_LSUI
11	0	1	LDTP	FEAT_LSUI
11	1	0	STTP (SIMD&FP)	FEAT_FP && FEAT_LSUI
11	1	1	LDTP (SIMD&FP)	FEAT_FP && FEAT_LSUI

Load/store register pair (offset)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	VR	0 1 0	L	imm7	Rt2	Rn	Rt

opc	VR	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	VR	L	Instruction Details	Feature
11	0	0	STTP	FEAT_LSUI
11	0	1	LDTP	FEAT_LSUI
11	1	0	STTP (SIMD&FP)	FEAT_FP && FEAT_LSUI
11	1	1	LDTP (SIMD&FP)	FEAT_FP && FEAT_LSUI

Load/store register pair (pre-indexed)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	VR	0 1 1	L	imm7	Rt2	Rn	Rt

opc	VR	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP
11	0	0	STTP	FEAT_LSUI
11	0	1	LDTP	FEAT_LSUI
11	1	0	STTP (SIMD&FP)	FEAT_FP && FEAT_LSUI
11	1	1	LDTP (SIMD&FP)	FEAT_FP && FEAT_LSUI

Load/store register (unscaled immediate)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	0	imm9	0 0	Rn	Rt

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Feature
00	0	00	STURB	-
00	0	01	LDURB	-
00	0	10	LDURSB — 64-bit	-
00	0	11	LDURSB — 32-bit	-
00	1	00	STUR (SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDUR (SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STUR (SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDUR (SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STURH	-
01	0	01	LDURH	-
01	0	10	LDURSH — 64-bit	-
01	0	11	LDURSH — 32-bit	-
01	1	00	STUR (SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDUR (SIMD&FP) — 16-bit	FEAT_FP
1x	0	11	UNALLOCATED	-
10	0	00	STUR — 32-bit	-
10	0	01	LDUR — 32-bit	-
10	0	10	LDURSW	-
10	1	00	STUR (SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDUR (SIMD&FP) — 32-bit	FEAT_FP
11	0	00	STUR — 64-bit	-
11	0	01	LDUR — 64-bit	-
11	0	10	PRFUM	-
11	1	00	STUR (SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDUR (SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x	UNALLOCATED	-

Load/store register (immediate post-indexed)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	0	imm9	0 1	Rn	Rt

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Feature
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	1x	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-
10	0	10	LDRSW (immediate)	-
10	0	11	UNALLOCATED	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	1x	UNALLOCATED	-
11	x	1x	UNALLOCATED	-
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP

Load/store register (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	0	imm9	1 0	Rn	Rt

size	VR	opc	Instruction Details
xx	1	xx	UNALLOCATED

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details
00	0	00	STTRB
00	0	01	LDTRB
00	0	10	LDTRSB — 64-bit
00	0	11	LDTRSB — 32-bit
01	0	00	STTRH
01	0	01	LDTRH
01	0	10	LDTRSH — 64-bit
01	0	11	LDTRSH — 32-bit
10	0	00	STTR — 32-bit
10	0	01	LDTR — 32-bit
10	0	10	LDTRSW
10	0	11	UNALLOCATED
11	0	00	STTR — 64-bit
11	0	01	LDTR — 64-bit
11	0	1x	UNALLOCATED

Load/store register (immediate pre-indexed)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	0	imm9	1 1	Rn	Rt

size	VR	opc	Instruction Details	Feature
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Feature
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	1x	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-
10	0	10	LDRSW (immediate)	-
10	0	11	UNALLOCATED	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	1x	UNALLOCATED	-
11	x	1x	UNALLOCATED	-
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP

Atomic memory operations

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	A	R	1	Rs	o3	opc	0 0	Rn	Rt

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
xx	0	0	x	xxxxx	1	100	xxxxx	UNALLOCATED	-
xx	0	0	x	xxxxx	1	11x	xxxxx	UNALLOCATED	-
xx	0	1	1	xxxxx	1	100	xxxxx	UNALLOCATED	-
xx	1	x	x	xxxxx	0	001	xxxxx	UNALLOCATED	-
xx	1	x	x	xxxxx	0	01x	xxxxx	UNALLOCATED	-
xx	1	0	x	xxxxx	1	xxx	!= 11111	UNALLOCATED	-
xx	1	0	x	xxxxx	1	001	11111	UNALLOCATED	-
xx	1	0	x	xxxxx	1	01x	11111	UNALLOCATED	-
xx	1	1	x	xxxxx	1	xxx	xxxxx	UNALLOCATED	-
0x	0	x	0	xxxxx	1	101	xxxxx	UNALLOCATED	-
0x	0	0	1	xxxxx	1	101	xxxxx	UNALLOCATED	-
0x	0	1	x	xxxxx	1	11x	xxxxx	UNALLOCATED	-
0x	0	1	1	xxxxx	1	101	xxxxx	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
00	0	0	0	xxxxx	0	000	xxxxx	LDADDB, LDADDAB, LDADDALB, LDADDLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	001	xxxxx	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	010	xxxxx	LDEORB, LDEORAB, LDEORALB, LDEORLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	011	xxxxx	LDSETB, LDSETAB, LDSETALB, LDSETLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	100	xxxxx	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	101	xxxxx	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	110	xxxxx	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	111	xxxxx	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	1	000	xxxxx	SWPB, SWPAB, SWPALB, SWPLB — SWPB	FEAT_LSE
00	0	0	0	xxxxx	1	001	xxxxx	RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL — RCWCLR	FEAT_THE
00	0	0	0	xxxxx	1	010	xxxxx	RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL — RCWSWP	FEAT_THE
00	0	0	0	xxxxx	1	011	xxxxx	RCWSET, RCWSETA, RCWSETAL, RCWSETL — RCWSET	FEAT_THE
00	0	0	1	xxxxx	0	000	xxxxx	LDADDB, LDADDAB, LDADDALB, LDADDLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	001	xxxxx	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	010	xxxxx	LDEORB, LDEORAB, LDEORALB, LDEORLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	011	xxxxx	LDSETB, LDSETAB, LDSETALB, LDSETLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	100	xxxxx	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	101	xxxxx	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	110	xxxxx	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	111	xxxxx	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Release	FEAT_LSE
00	0	0	1	xxxxx	1	000	xxxxx	SWPB, SWPAB, SWPALB, SWPLB — SWPLB	FEAT_LSE
00	0	0	1	xxxxx	1	001	xxxxx	RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL — RCWCLRL	FEAT_THE
00	0	0	1	xxxxx	1	010	xxxxx	RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL — RCWSWPL	FEAT_THE
00	0	0	1	xxxxx	1	011	xxxxx	RCWSET, RCWSETA, RCWSETAL, RCWSETL — RCWSETL	FEAT_THE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
00	0	1	0	xxxxx	0	000	xxxxx	LDADDB, LDADDAB, LDADDALB, LDADDLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	001	xxxxx	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	010	xxxxx	LDEORB, LDEORAB, LDEORALB, LDEORLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	011	xxxxx	LDSETB, LDSETAB, LDSETALB, LDSETLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	100	xxxxx	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	101	xxxxx	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	110	xxxxx	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	111	xxxxx	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	1	000	xxxxx	SWPB, SWPAB, SWPALB, SWPLB — SWPAB	FEAT_LSE
00	0	1	0	xxxxx	1	001	xxxxx	RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL — RCWCLRA	FEAT_THE
00	0	1	0	xxxxx	1	010	xxxxx	RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL — RCWSWPA	FEAT_THE
00	0	1	0	xxxxx	1	011	xxxxx	RCWSET, RCWSETA, RCWSETAL, RCWSETL — RCWSETA	FEAT_THE
00	0	1	0	xxxxx	1	100	xxxxx	LDAPRB	FEAT_LRCPC
00	0	1	1	xxxxx	0	000	xxxxx	LDADDB, LDADDAB, LDADDALB, LDADDLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	001	xxxxx	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	010	xxxxx	LDEORB, LDEORAB, LDEORALB, LDEORLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	011	xxxxx	LDSETB, LDSETAB, LDSETALB, LDSETLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	100	xxxxx	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	101	xxxxx	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	110	xxxxx	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	111	xxxxx	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	1	000	xxxxx	SWPB, SWPAB, SWPALB, SWPLB — SWPALB	FEAT_LSE
00	0	1	1	xxxxx	1	001	xxxxx	RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL — RCWCLRAL	FEAT_THE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
00	0	1	1	xxxxx	1	010	xxxxx	RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL — RCWSWPAL	FEAT_THE
00	0	1	1	xxxxx	1	011	xxxxx	RCWSET, RCWSETA, RCWSETAL, RCWSETL — RCWSETAL	FEAT_THE
00	1	0	0	xxxxx	0	000	xxxxx	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	0	100	xxxxx	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	0	101	xxxxx	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	0	110	xxxxx	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	0	111	xxxxx	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	000	11111	STBFADD, STBFADDL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	100	11111	STBFMAX, STBFMAXL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	101	11111	STBFMIN, STBFMINL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	110	11111	STBFMAXNM, STBFMAXNML — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	111	11111	STBFMINNM, STBFMINNML — No memory ordering	FEAT_LSFE
00	1	0	1	xxxxx	0	000	xxxxx	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL — Release	FEAT_LSFE
00	1	0	1	xxxxx	0	100	xxxxx	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL — Release	FEAT_LSFE
00	1	0	1	xxxxx	0	101	xxxxx	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL — Release	FEAT_LSFE
00	1	0	1	xxxxx	0	110	xxxxx	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML — Release	FEAT_LSFE
00	1	0	1	xxxxx	0	111	xxxxx	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	000	11111	STBFADD, STBFADDL — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	100	11111	STBFMAX, STBFMAXL — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	101	11111	STBFMIN, STBFMINL — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	110	11111	STBFMAXNM, STBFMAXNML — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	111	11111	STBFMINNM, STBFMINNML — Release	FEAT_LSFE
00	1	1	0	xxxxx	0	000	xxxxx	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL — Acquire	FEAT_LSFE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
00	1	1	0	xxxxx	0	100	xxxxx	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL — Acquire	FEAT_LSFE
00	1	1	0	xxxxx	0	101	xxxxx	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL — Acquire	FEAT_LSFE
00	1	1	0	xxxxx	0	110	xxxxx	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML — Acquire	FEAT_LSFE
00	1	1	0	xxxxx	0	111	xxxxx	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML — Acquire	FEAT_LSFE
00	1	1	1	xxxxx	0	000	xxxxx	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL — Acquire-release	FEAT_LSFE
00	1	1	1	xxxxx	0	100	xxxxx	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL — Acquire-release	FEAT_LSFE
00	1	1	1	xxxxx	0	101	xxxxx	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL — Acquire-release	FEAT_LSFE
00	1	1	1	xxxxx	0	110	xxxxx	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML — Acquire-release	FEAT_LSFE
00	1	1	1	xxxxx	0	111	xxxxx	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML — Acquire-release	FEAT_LSFE
01	0	0	0	xxxxx	0	000	xxxxx	LDADDH, LDADDAH, LDADDALH, LDADDLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	001	xxxxx	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	010	xxxxx	LDEORH, LDEORAH, LDEORALH, LDEORLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	011	xxxxx	LDSETH, LDSETAH, LDSETALH, LDSETLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	100	xxxxx	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	101	xxxxx	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	110	xxxxx	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	111	xxxxx	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	1	000	xxxxx	SWPH, SWPAH, SWPALH, SWPLH — SWPH	FEAT_LSE
01	0	0	0	xxxxx	1	001	xxxxx	RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL — RCWSCLR	FEAT_THE
01	0	0	0	xxxxx	1	010	xxxxx	RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL — RCWSSWP	FEAT_THE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
01	0	0	0	xxxxx	1	011	xxxxx	RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL — RCWSSET	FEAT_THE
01	0	0	1	xxxxx	0	000	xxxxx	LDADDH, LDADDAH, LDADDALH, LDADDLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	001	xxxxx	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	010	xxxxx	LDEORH, LDEORAH, LDEORALH, LDEORLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	011	xxxxx	LDSETH, LDSETAH, LDSETALH, LDSETLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	100	xxxxx	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	101	xxxxx	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	110	xxxxx	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	111	xxxxx	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Release	FEAT_LSE
01	0	0	1	xxxxx	1	000	xxxxx	SWPH, SWPAH, SWPALH, SWPLH — SWPLH	FEAT_LSE
01	0	0	1	xxxxx	1	001	xxxxx	RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL — RCWSCLRL	FEAT_THE
01	0	0	1	xxxxx	1	010	xxxxx	RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL — RCWSSWPL	FEAT_THE
01	0	0	1	xxxxx	1	011	xxxxx	RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL — RCWSSETL	FEAT_THE
01	0	1	0	xxxxx	0	000	xxxxx	LDADDH, LDADDAH, LDADDALH, LDADDLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	001	xxxxx	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	010	xxxxx	LDEORH, LDEORAH, LDEORALH, LDEORLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	011	xxxxx	LDSETH, LDSETAH, LDSETALH, LDSETLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	100	xxxxx	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	101	xxxxx	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	110	xxxxx	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	111	xxxxx	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	1	000	xxxxx	SWPH, SWPAH, SWPALH, SWPLH — SWPAH	FEAT_LSE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
01	0	1	0	xxxxx	1	001	xxxxx	RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL — RCWSCLRA	FEAT_THE
01	0	1	0	xxxxx	1	010	xxxxx	RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL — RCWSSWPA	FEAT_THE
01	0	1	0	xxxxx	1	011	xxxxx	RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL — RCWSSETA	FEAT_THE
01	0	1	0	xxxxx	1	100	xxxxx	LDAPRH	FEAT_LRCPC
01	0	1	1	xxxxx	0	000	xxxxx	LDADDH, LDADDAH, LDADDALH, LDADDLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	001	xxxxx	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	010	xxxxx	LDEORH, LDEORAH, LDEORALH, LDEORLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	011	xxxxx	LDSETH, LDSETAH, LDSETALH, LDSETLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	100	xxxxx	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	101	xxxxx	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	110	xxxxx	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	111	xxxxx	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	1	000	xxxxx	SWPH, SWPAH, SWPALH, SWPLH — SWPALH	FEAT_LSE
01	0	1	1	xxxxx	1	001	xxxxx	RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL — RCWSCLRAL	FEAT_THE
01	0	1	1	xxxxx	1	010	xxxxx	RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL — RCWSSWPAL	FEAT_THE
01	0	1	1	xxxxx	1	011	xxxxx	RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL — RCWSSETAL	FEAT_THE
01	1	0	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Half-precision no memory ordering	FEAT_LSFE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
01	1	0	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	000	11111	STFADD, STFADDL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	100	11111	STFMAX, STFMAXL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	101	11111	STFMIN, STFMINL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	111	11111	STFMINNM, STFMINNML — Half-precision no memory ordering	FEAT_LSFE
01	1	0	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	000	11111	STFADD, STFADDL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	100	11111	STFMAX, STFMAXL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	101	11111	STFMIN, STFMINL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	111	11111	STFMINNM, STFMINNML — Half-precision release	FEAT_LSFE
01	1	1	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Half-precision acquire	FEAT_LSFE
01	1	1	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Half-precision acquire	FEAT_LSFE
01	1	1	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Half-precision acquire	FEAT_LSFE
01	1	1	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Half-precision acquire	FEAT_LSFE
01	1	1	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Half-precision acquire	FEAT_LSFE
01	1	1	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Half-precision acquire-release	FEAT_LSFE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
01	1	1	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Half-precision acquire-release	FEAT_LSFE
01	1	1	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Half-precision acquire-release	FEAT_LSFE
01	1	1	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Half-precision acquire-release	FEAT_LSFE
01	1	1	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Half-precision acquire-release	FEAT_LSFE
1x	0	1	x	xxxxx	1	x01	xxxxx	UNALLOCATED	-
1x	0	1	x	xxxxx	1	x1x	xxxxx	UNALLOCATED	-
10	0	0	x	xxxxx	1	x01	xxxxx	UNALLOCATED	-
10	0	0	x	xxxxx	1	010	xxxxx	UNALLOCATED	-
10	0	0	x	xxxxx	1	011	xxxxx	UNALLOCATED	-
10	0	0	0	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 32-bit SWP	FEAT_LSE
10	0	0	1	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit release	FEAT_LSE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
10	0	0	1	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 32-bit SWPL	FEAT_LSE
10	0	1	0	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 32-bit SWPA	FEAT_LSE
10	0	1	0	xxxxx	1	100	xxxxx	LDAPR — 32-bit	FEAT_LRCPC
10	0	1	1	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 32-bit SWPAL	FEAT_LSE
10	1	0	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Single-precision no memory ordering	FEAT_LSFE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
10	1	0	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	000	11111	STFADD, STFADDL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	100	11111	STFMAX, STFMAXL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	101	11111	STFMIN, STFMINL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	111	11111	STFMINNM, STFMINNML — Single-precision no memory ordering	FEAT_LSFE
10	1	0	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	000	11111	STFADD, STFADDL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	100	11111	STFMAX, STFMAXL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	101	11111	STFMIN, STFMINL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	111	11111	STFMINNM, STFMINNML — Single-precision release	FEAT_LSFE
10	1	1	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Single-precision acquire	FEAT_LSFE
10	1	1	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Single-precision acquire	FEAT_LSFE
10	1	1	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Single-precision acquire	FEAT_LSFE
10	1	1	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Single-precision acquire	FEAT_LSFE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
10	1	1	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Single-precision acquire	FEAT_LSFE
10	1	1	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Single-precision acquire-release	FEAT_LSFE
10	1	1	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Single-precision acquire-release	FEAT_LSFE
10	1	1	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Single-precision acquire-release	FEAT_LSFE
10	1	1	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Single-precision acquire-release	FEAT_LSFE
10	1	1	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Single-precision acquire-release	FEAT_LSFE
11	0	0	0	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 64-bit SWP	FEAT_LSE
11	0	0	0	xxxxx	1	010	xxxxx	ST64BV0	FEAT_LS64_ACCDATA
11	0	0	0	xxxxx	1	011	xxxxx	ST64BV	FEAT_LS64_V
11	0	0	0	11111	1	001	xxxxx	ST64B	FEAT_LS64
11	0	0	0	11111	1	101	xxxxx	LD64B	FEAT_LS64
11	0	0	0	!= 11111	1	x01	xxxxx	UNALLOCATED	-
11	0	0	1	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit release	FEAT_LSE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
11	0	0	1	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	1	x01	xxxxx	UNALLOCATED	-
11	0	0	1	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 64-bit SWPL	FEAT_LSE
11	0	0	1	xxxxx	1	010	xxxxx	UNALLOCATED	-
11	0	0	1	xxxxx	1	011	xxxxx	UNALLOCATED	-
11	0	1	0	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 64-bit SWPA	FEAT_LSE
11	0	1	0	xxxxx	1	100	xxxxx	LDAPR — 64-bit	FEAT_LRCPC
11	0	1	1	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit acquire-release	FEAT_LSE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
11	0	1	1	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 64-bit SWPAL	FEAT_LSE
11	1	0	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	000	11111	STFADD, STFADDL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	100	11111	STFMAX, STFMAXL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	101	11111	STFMIN, STFMINL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	111	11111	STFMINNM, STFMINNML — Double-precision no memory ordering	FEAT_LSFE
11	1	0	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	000	11111	STFADD, STFADDL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	100	11111	STFMAX, STFMAXL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	101	11111	STFMIN, STFMINL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	111	11111	STFMINNM, STFMINNML — Double-precision release	FEAT_LSFE

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Feature
11	1	1	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Double-precision acquire	FEAT_LSFE
11	1	1	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Double-precision acquire	FEAT_LSFE
11	1	1	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Double-precision acquire	FEAT_LSFE
11	1	1	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Double-precision acquire	FEAT_LSFE
11	1	1	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Double-precision acquire	FEAT_LSFE
11	1	1	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Double-precision acquire-release	FEAT_LSFE
11	1	1	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Double-precision acquire-release	FEAT_LSFE
11	1	1	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Double-precision acquire-release	FEAT_LSFE
11	1	1	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Double-precision acquire-release	FEAT_LSFE
11	1	1	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Double-precision acquire-release	FEAT_LSFE

Load/store register (register offset)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	1	Rm	option	S	1 0	Rn	Rt

size	VR	opc	option	Rt	Instruction Details	Feature
00	0	00	011	xxxxx	STRB (register) — Shifted register	-
00	0	00	!= 011	xxxxx	STRB (register) — Extended register	-
00	0	01	011	xxxxx	LDRB (register) — Shifted register	-
00	0	01	!= 011	xxxxx	LDRB (register) — Extended register	-
00	0	10	011	xxxxx	LDRSB (register) — 64-bit with shifted register offset	-
00	0	10	!= 011	xxxxx	LDRSB (register) — 64-bit with extended register offset	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	option	Rt	Instruction Details	Feature
00	0	11	011	xxxxx	LDRSB (register) — 32-bit with shifted register offset	-
00	0	11	!= 011	xxxxx	LDRSB (register) — 32-bit with extended register offset	-
00	1	00	011	xxxxx	STR (register, SIMD&FP) — 8-bit with shifted register offset	FEAT_FP
00	1	00	!= 011	xxxxx	STR (register, SIMD&FP) — 8-bit with extended register offset	FEAT_FP
00	1	01	011	xxxxx	LDR (register, SIMD&FP) — 8-bit with shifted register offset	FEAT_FP
00	1	01	!= 011	xxxxx	LDR (register, SIMD&FP) — 8-bit with extended register offset	FEAT_FP
00	1	10	xxx	xxxxx	STR (register, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	xxx	xxxxx	LDR (register, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	xxx	xxxxx	STRH (register)	-
01	0	01	xxx	xxxxx	LDRH (register)	-
01	0	10	xxx	xxxxx	LDRSH (register) — 64-bit	-
01	0	11	xxx	xxxxx	LDRSH (register) — 32-bit	-
01	1	00	xxx	xxxxx	STR (register, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	xxx	xxxxx	LDR (register, SIMD&FP) — 16-bit	FEAT_FP
1x	0	11	xxx	xxxxx	UNALLOCATED	-
10	0	00	xxx	xxxxx	STR (register) — 32-bit	-
10	0	01	xxx	xxxxx	LDR (register) — 32-bit	-
10	0	10	xxx	xxxxx	LDRSW (register)	-
10	1	00	xxx	xxxxx	STR (register, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	xxx	xxxxx	LDR (register, SIMD&FP) — 32-bit	FEAT_FP
11	0	00	xxx	xxxxx	STR (register) — 64-bit	-
11	0	01	xxx	xxxxx	LDR (register) — 64-bit	-
11	0	10	x0x	xxxxx	UNALLOCATED	-
11	0	10	x1x	11xxx	RPRFM	FEAT_RPRFM
11	0	10	x1x	!= 11xxx	PRFM (register)	-
11	1	00	xxx	xxxxx	STR (register, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	xxx	xxxxx	LDR (register, SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x	xxx	xxxxx	UNALLOCATED	-

Load/store register (pac)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23	22	21	20 12	11	10	9 5	4 0
size	1 1 1	VR	0 0	M	S	1	imm9	W	1	Rn	Rt

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	M	W	Instruction Details	Feature
11	0	0	0	LDRAA, LDRAB — Key A, offset	FEAT_PAuth
11	0	0	1	LDRAA, LDRAB — Key A, pre-indexed	FEAT_PAuth
11	0	1	0	LDRAA, LDRAB — Key B, offset	FEAT_PAuth
11	0	1	1	LDRAA, LDRAB — Key B, pre-indexed	FEAT_PAuth
11	1	x	x	UNALLOCATED	-
!=11	x	x	x	UNALLOCATED	-

Load/store register (unsigned immediate)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
size	1 1 1	VR	0 1	opc	imm12	Rn	Rt

size	VR	opc	Instruction Details	Feature
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
1x	0	11	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-
10	0	10	LDRSW (immediate)	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Feature
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	0	10	PRFM (immediate)	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x	UNALLOCATED	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C5
The A64 System Instruction Class

This chapter describes the A64 System instruction class, and the System instruction class encoding space, that is a subset of the System registers encoding space. It contains the following sections:

- The System instruction class encoding space.
- Special purpose registers.
- A64 System instructions for cache maintenance.
- A64 System instructions for address translation.
- A64 System instructions for TLB maintenance.
- A64 System instructions for prediction restriction.
- A64 System instructions for the Branch Record Buffer Extension.
- A64 System instructions for the Trace Extension.
- A64 System instructions for the Guarded Control Stack.
- A64 IMPLEMENTATION DEFINED System instructions.

See General information about the A64 instruction descriptions for information about entries used in the instruction encoding descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.1 The System instruction class encoding space

Part of the A64 instruction encoding space is assigned to instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the `CLREX` instruction.
- Architectural hint instructions.

This section describes the general model for accessing this functionality.

Note

- See Fixed values in AArch64 instruction and System register descriptions for information about abbreviations used in the System instruction descriptions.
- In AArch32 state much of this functionality is provided through the System register interface described in The AArch32 System register interface. In AArch64 state, the parameters used to characterize the System register encoding space are {`op0`, `op1`, `CRn`, `CRm`, `op2`}. These are based on the parameters that characterize the AArch32 System register encoding space, which reflect the original implementation of these registers, as described in Background to the System register interface. There is no particular significance to the naming of these parameters, and no functional distinction between the `opn` parameters and the `CRx` parameters.

Principles of the System instruction class encoding describes some general properties of these encodings. System instruction class encoding overview then describes the top-level encoding of these instructions, and the following sections then describe the next level of the encoding hierarchy of System instructions and Special-purpose registers:

- op0 == `0b00`, architectural hints, barriers and CLREX, and PSTATE access.
- op0 == `0b01`, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions.
- op0 == `0b11`, Moves to and from Special-purpose registers.

For the description of the next level of encoding hierarchy of System registers, see:

- Moves to and from debug and trace System registers.
- Moves to and from non-debug System registers, Special-purpose registers.
- Reserved encodings for IMPLEMENTATION DEFINED registers.

C5.1.1 Principles of the System instruction class encoding

An encoding in the System instruction space is identified by a set of arguments,{`op0`, `op1`, `CRn`, `CRm`, `op2`}. These form an encoding hierarchy, where:

`op0` Defines the top-level division of the encoding space, see System instruction class encoding overview.

`op1` Identifies the lowest Exception level at which the encoding is accessible, as follows:

Accessible at EL0 `op1` has the value `0b011`.

Accessible at EL1 `op1` has the value `0b000`, `0b001`, or `0b010`. The value is the same as the `op1` value used to access the equivalent AArch32 register.

Accessible at Secure EL1 `op1` has the value `0b111`.

Accessible at EL2 `op1` has the value `0b100` or `0b101`. The value `0b101` is used for the EL12 encodings that access EL1 System registers used when FEAT_VHE is implemented and the Effective value of HCR_EL2.E2H is 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible at EL3 `op1` has the value `0b110`.

Arm strongly recommends that implementers adopt this use of `op1` when using the IMPLEMENTATION DEFINED regions of the encoding space described in Reserved encodings for IMPLEMENTATION DEFINED registers.

C5.1.2 System instruction class encoding overview

The encoding of the System instruction class describes each instruction as being either:

- A transfer to a System register. This is a System instruction with the semantics of a write.
- A transfer from a System register. This is a System instruction with the semantics of a read.

A System instruction that initiates an operation operates as if it was making a transfer to a register.

In the AArch64 instruction set, the decode structure for the System instruction class is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	op0	op1	CRn	CRm	op2	Rt

The value of `L` indicates the transfer direction:

0 Transfer to System register.

1 Transfer from System register.

The `op0` field is the top-level encoding of the System instruction type. Its possible values are:

`0b00` These encodings provide:

- Instructions with an immediate field for accessing PSTATE, the current PE state.
- The architectural hint instructions.
- Barriers and the `CLREX` instruction.

For more information about these encodings, see op0 == `0b00`, architectural hints, barriers and CLREX, and PSTATE access.

`0b01` These encodings provide:

- Cache maintenance instructions.
- TLB maintenance instructions.
- Address translation instructions.
- Prediction restriction instructions.
- BRBE instructions.

Note

These are equivalent to operations in the AArch32 (`coproc` == `0b1111`) encoding space.

For more information, see op0 == `0b01`, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions.

`0b10` These encodings provide moves to and from:

- Legacy AArch32 System registers for execution environments, to provide access to these registers from higher Exception levels that are using AArch64.
- Debug, PMU, System PMU, and trace registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

These are equivalent to the registers in the AArch32 (`coproc` == `0b1110`) encoding space.

For more information, see Moves to and from debug and trace System registers.

`0b11` These encodings provide:

- Moves to and from Non-debug System registers. The accessed registers provide system control, and system status information.

  Note

  The accessed registers are equivalent to the registers in the AArch32 (`coproc` == `0b1111`) encoding space.

- Access to Special-purpose registers.

For more information, see Instructions for accessing Special-purpose registers and Instructions for accessing non-debug System registers.

C5.1.2.1 UNDEFINED behaviors

In the System register instruction encoding space, the following principles apply:

- All unallocated encodings are treated as UNDEFINED.
- All encodings with `L` == 1 and `op0` == `0b0x` are UNDEFINED, except for encodings in the area reserved for IMPLEMENTATION DEFINED use, see Reserved encoding space for IMPLEMENTATION DEFINED instructions.

For registers and operations that are accessible from a particular Exception level, any attempt to access those registers from a lower Exception level is UNDEFINED.

If a particular Exception level:

- Defines a register to be RO, then any attempt to write to that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L` == 0 is UNDEFINED.
- Defines a register to be WO, then any attempt to read from that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L` == 1 is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED, but see the recommendation in Principles of the System instruction class encoding.

C5.1.3 `op0`== `0b00`, architectural hints, barriers and `CLREX`, and PSTATE access

The different groups of System register instructions with `op0`== `0b00`:

- Are identified by the value of `CRn`.
- Are always encoded with a value of `0b11111` in the `Rt` field.

The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4	0
1 1 0 1	0 1 0 1	0 0	L	0 0	op1	CRn	CRm	op2	1 1 1 1 1	
				op0						Rt

The encoding of the `CRn` field is as follows:

`0b0010` See Architectural hint instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b0011` See Barriers and CLREX.

`0b0100` See Instructions for accessing the PSTATE fields.

C5.1.3.1 Architectural hint instructions

Within the `op0==0b00` encodings, the architectural hint instructions are identified by `CRn` having the value `0b0010`. The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	5	4	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1 1	0	0 1 0	Op<6:0>		1	1 1 1 1
											op0		op1		CRn		CRm	op2		Rt

The value of `op<6:0>`, formed by concatenating the `CRm` and `op2` fields, determines the hint instruction as follows:

`0b0000000` `NOP` instruction.

`0b0000001` `YIELD` instruction.

`0b0000010` `WFE` instruction.

`0b0000011` `WFI` instruction.

`0b0000100` `SEV` instruction.

`0b0000101` `SEVL` instruction.

`0b0000110` `DGH` instruction.

`0b0000111` `XPACD, XPACI, XPACLRI` instruction.

`0b0001000` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIA1716 variant.

`0b0001010` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIB1716 variant.

`0b0001100` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIA1716 variant.

`0b0001110` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIB1716 variant.

`0b0010000` `ESB` instruction.

`0b0010001`

: `PSB` instruction.

`0b0010010`

: `TSB` instruction.

`0b0010100` `CSDB` instruction.

`0b0010110` `CLRBHB` instruction.

`0b0011000` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIAZ variant.

`0b0011001` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIASP variant.

`0b0011010` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIBZ variant.

`0b0011011` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIBSP variant.

`0b0011100` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIAZ variant.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b0011101` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIASP variant.

`0b0011110` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIBZ variant.

`0b0011111` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIBSP variant.

`0b0101000` `CHKFEAT` instruction.

`0b0100xx0` `BTI` instruction.

`0b011000x` `STSHH` instruction.

These instructions are described in A64 Base Instruction Descriptions.

Note

- Instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.
- The operation of the A64 instructions for architectural hints are identical to the corresponding T32 and A32 instructions.

For more information about:

- The `WFE`, `WFI`, `SEV`, and `SEVL` instructions, see Mechanisms for entering a low-power state.
- The `YIELD` instruction, see Software control features and EL0.
- The `STSHH` instruction, see Use of the FEAT_PCDPHINT STSHH and PRFM IR (immediate) instructions.

C5.1.3.2 Barriers and `CLREX`

Within the `op0==0b00` encodings, the barriers and `CLREX` instructions are identified by `CRn` having the value `0b0011`. The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22 21	20	19	18	16	15	12	11	8	7	5	4	0
1 1 0 1	0 1 0 1	0 0 0	0	0	0 1	1	0 0 1	1	CRm		op2		1 1 1 1	1
			op0		op1		CRn						Rt	

The value of `op2` determines the instruction, as follows.

`0b001` `DSB` instruction, Memory nXS barrier variant.

`0b010` `CLREX` instruction.

`0b100` `DSB` instruction, Memory barrier variant.

`0b101` `DMB` instruction.

`0b110` `ISB` instruction.

`0b111` `SB` instruction.

`0b000`; `0b011` UNDEFINED.

These instructions are described in A64 Base Instruction Descriptions.

Note

- Instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.
- The operation of the A64 instructions for barriers and `CLREX` are identical to the corresponding T32 and A32 instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about:

- The barrier instructions, see Memory barriers.
- The `CLREX` instruction, see Synchronization and semaphores.

C5.1.3.3 Instructions for accessing the PSTATE fields

Within the `op0==0b00` encodings, the instructions that can be used to modify PSTATE fields directly are identified by `CRn` having the value `0b0100`. The encoding of these instructions is shown in Figure C5-1 and Figure C5-2:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15	14	13	12	11 … 8	7 … 5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1	0	1	0	0	Imm4	op2	1	1	1	1	1
											op0			CRn				CRm		Rt				

Figure C5-1 Instructions using #Imm4

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15	14	13	12	11	10	9	8	7 … 5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1	0	1	0	0	0	0	0	Imm1	op2	1	1	1	1	1
											op0			CRn				CRm					Rt				

Figure C5-2 Instructions using #Imm1

These instructions are:

```
CFINV                 ; Inverts the value of PSTATE.C
MSR DAIFSet, #Imm4    ; Used to set any or all of DAIF to 1
MSR DAIFClr, #Imm4    ; Used to clear any or all of DAIF to 0
MSR SPSel, #Imm4      ; Used to select the Stack Pointer, between SP_EL0 and SP_ELx
MSR UAO, #Imm4        ; Used to set the value of PSTATE.UAO
MSR PAN, #Imm4        ; Used to set the value of PSTATE.PAN
MSR DIT, #Imm4        ; Used to set the value of PSTATE.DIT
MSR SSBS, #Imm4       ; Used to set the value of PSTATE.SSBS
MSR TCO, #Imm4        ; Used to set the value of PSTATE.TCO
MSR ALLINT, #Imm1     ; Used to set the value of PSTATE.ALLINT
```

The value of `op2` selects the instruction form, which defines the constraints on the values of the `op1` and `Imm4` or `Imm1` arguments, as follows:

`op2 == 0b000` Selects the `CFINV` instruction.

`op2 == 0b011` Selects the `MSR UAO` instruction.

`op2 == 0b100` Selects the `MSR PAN` instruction.

`op2 == 0b101` Selects the `MSR SPSel` instruction.

`op2 == 0b001` Selects the `MSR SSBS` instruction.

`op2 == 0b010` Selects the `MSR DIT` instruction.

`op2 == 0b100` Selects the `MSR TCO` instruction.

`op2 == 0b110` Selects the `MSR DAIFSet` instruction, that sets the specified PSTATE.{D, A, I, F} bits to 1.

`op2 == 0b111` Selects the `MSR DAIFClr` instruction, that clears the specified PSTATE.{D, A, I, F} bits to 0.

`op2 == 0b000` Selects the `MSR ALLINT` instruction.

All other combinations of `op1` and `op2` are reserved, and the corresponding instructions are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For PSTATE updates, instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.

Writes to PSTATE occur in program order without the need for additional synchronization. Changing PSTATE.SP to use SP_EL0 synchronizes any updates to SP_EL0 that have been written by an `MSR` to SP_EL0, without the need for additional synchronization.

C5.1.4 `op0`== `0b01`, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions

The System instructions are encoded with `op0`== `0b01`. The different groups of System instructions are identified by the values of `CRn` and `CRm`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 0 0	1 op1	CRn	CRm	op2	Xt
			op0				

The grouping of these instructions depending on the `CRn` and `CRm` fields is as follows:

`CRn` == `0b0111` The instruction group is determined by the value of `CRm`, as follows:

`CRm` == {`0b0001`, `0b0101`} Instruction cache maintenance instructions.

See Cache maintenance instructions, and data cache zero operations.

`CRm` == `0b0010` Branch Record Buffer and Trace Extension instructions.

See Branch Record Buffer instructions and Trace Extension instructions.

`CRm` == `0b0011` Prediction restriction instructions.

See Prediction restriction instructions.

`CRm` == `0b0100` Data cache zero operations.

See Cache maintenance instructions, and data cache zero operations.

`CRm` == {`0b0110`, `0b1010`, `0b1011`, `0b1100`, `0b1101`, `0b1110`, `0b1111`} Data cache maintenance instructions.

See Cache maintenance instructions, and data cache zero operations.

`CRm` == `0b0111` Guarded Control Stack instructions.

See Guarded Control Stack instructions.

`CRm` == {`0b1000`, `0b1001`} See Address translation instructions.

`CRn` == {`0b1000`, `0b1001`} See TLB maintenance instructions.

`CRn` == {`0b1011`, `0b1111`} See Reserved encoding space for IMPLEMENTATION DEFINED instructions.

C5.1.4.1 Cache maintenance instructions, and data cache zero operations

Table C5-1 lists the Cache maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`. For these instructions, if the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction behaves as if the Xt field is set to `0b11111`.

Table C5-1 Cache maintenance instructions and data cache zero operations

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b0001	0b000	IC IALLUIS
0b01	0b000	0b0111	0b0101	0b000	IC IALLU
0b01	0b000	0b0111	0b0110	0b001	DC IVAC
0b01	0b000	0b0111	0b0110	0b010	DC ISW
0b01	0b000	0b0111	0b0110	0b011	DC IGVAC
0b01	0b000	0b0111	0b0110	0b100	DC IGSW
0b01	0b000	0b0111	0b0110	0b101	DC IGDVAC
0b01	0b000	0b0111	0b0110	0b110	DC IGDSW
0b01	0b000	0b0111	0b1010	0b010	DC CSW
0b01	0b000	0b0111	0b1010	0b100	DC CGSW
0b01	0b000	0b0111	0b1010	0b110	DC CGDSW
0b01	0b000	0b0111	0b1110	0b010	DC CISW
0b01	0b000	0b0111	0b1110	0b100	DC CIGSW
0b01	0b000	0b0111	0b1110	0b110	DC CIGDSW
0b01	0b000	0b0111	0b1111	0b001	DC CIVAPS
0b01	0b000	0b0111	0b1111	0b101	DC CIGDVAPS
0b01	0b011	0b0111	0b0100	0b001	DC ZVA
0b01	0b011	0b0111	0b0100	0b011	DC GVA
0b01	0b011	0b0111	0b0100	0b100	DC GZVA
0b01	0b011	0b0111	0b0101	0b001	IC IVAU
0b01	0b011	0b0111	0b1010	0b001	DC CVAC
0b01	0b011	0b0111	0b1010	0b011	DC CGVAC
0b01	0b011	0b0111	0b1010	0b101	DC CGDVAC
0b01	0b011	0b0111	0b1011	0b000	DC CVAOC
0b01	0b011	0b0111	0b1011	0b001	DC CVAU
0b01	0b011	0b0111	0b1011	0b111	DC CGDVAOC
0b01	0b011	0b0111	0b1100	0b001	DC CVAP
0b01	0b011	0b0111	0b1100	0b011	DC CGVAP
0b01	0b011	0b0111	0b1100	0b101	DC CGDVAP
0b01	0b011	0b0111	0b1101	0b001	DC CVADP
0b01	0b011	0b0111	0b1101	0b011	DC CGVADP
0b01	0b011	0b0111	0b1101	0b101	DC CGDVADP
0b01	0b011	0b0111	0b1110	0b001	DC CIVAC
0b01	0b011	0b0111	0b1110	0b011	DC CIGVAC
0b01	0b011	0b0111	0b1110	0b101	DC CIGDVAC

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b011	0b0111	0b1111	0b000	DC CIVAOC
0b01	0b011	0b0111	0b1111	0b111	DC CIGDVAOC
0b01	0b100	0b0111	0b1110	0b000	DC CIPAE
0b01	0b100	0b0111	0b1110	0b111	DC CIGDPAE
0b01	0b110	0b0111	0b1110	0b001	DC CIPAPA
0b01	0b110	0b0111	0b1110	0b101	DC CIGDPAPA

For more information about these instructions, see About cache maintenance in AArch64 state and A64 Cache maintenance instructions.

C5.1.4.2 Prediction restriction instructions

Table C5-2 lists the Prediction restriction instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax.

Table C5-2 Prediction restriction instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b011	0b0111	0b0011	0b100	CFP RCTX
0b01	0b011	0b0111	0b0011	0b101	DVP RCTX
0b01	0b011	0b0111	0b0011	0b110	COSP RCTX
0b01	0b011	0b0111	0b0011	0b111	CPP RCTX

For more information about these instructions, see Execution, data prediction and prefetching restriction System instructions.

C5.1.4.3 Address translation instructions

Table C5-3 lists the Address translation instructions and their encodings. The syntax of the instructions includes `Xt`, that provides the address to be translated.

Table C5-3 Address translation instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b1000	0b000	AT S1E1R
0b01	0b000	0b0111	0b1000	0b001	AT S1E1W
0b01	0b000	0b0111	0b1000	0b010	AT S1E0R
0b01	0b000	0b0111	0b1000	0b011	AT S1E0W
0b01	0b000	0b0111	0b1001	0b000	AT S1E1RP
0b01	0b000	0b0111	0b1001	0b001	AT S1E1WP
0b01	0b000	0b0111	0b1001	0b010	AT S1E1A
0b01	0b100	0b0111	0b1000	0b000	AT S1E2R

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b100	0b0111	0b1000	0b001	AT S1E2W
0b01	0b100	0b0111	0b1000	0b100	AT S12E1R
0b01	0b100	0b0111	0b1000	0b101	AT S12E1W
0b01	0b100	0b0111	0b1000	0b110	AT S12E0R
0b01	0b100	0b0111	0b1000	0b111	AT S12E0W
0b01	0b100	0b0111	0b1001	0b010	AT S1E2A
0b01	0b110	0b0111	0b1000	0b000	AT S1E3R
0b01	0b110	0b0111	0b1000	0b001	AT S1E3W
0b01	0b110	0b0111	0b1001	0b010	AT S1E3A

For more information about these instructions, see Address translation instructions.

C5.1.4.4 TLB maintenance instructions

Table C5-4 lists the TLB maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`. For these instructions, if the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Table C5-4 TLB maintenance instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1000	0b0001	0b000	TLBI VMALLE1OS
0b01	0b000	0b1000	0b0001	0b001	TLBI VAE1OS
0b01	0b000	0b1000	0b0001	0b001	TLBIP VAE1OS
0b01	0b000	0b1000	0b0001	0b010	TLBI ASIDE1OS
0b01	0b000	0b1000	0b0001	0b011	TLBI VAAE1OS
0b01	0b000	0b1000	0b0001	0b011	TLBIP VAAE1OS
0b01	0b000	0b1000	0b0001	0b101	TLBI VALE1OS
0b01	0b000	0b1000	0b0001	0b101	TLBIP VALE1OS
0b01	0b000	0b1000	0b0001	0b111	TLBI VAALE1OS
0b01	0b000	0b1000	0b0001	0b111	TLBIP VAALE1OS
0b01	0b000	0b1000	0b0010	0b001	TLBI RVAE1IS
0b01	0b000	0b1000	0b0010	0b001	TLBIP RVAE1IS
0b01	0b000	0b1000	0b0010	0b011	TLBI RVAAE1IS
0b01	0b000	0b1000	0b0010	0b011	TLBIP RVAAE1IS
0b01	0b000	0b1000	0b0010	0b101	TLBI RVALE1IS
0b01	0b000	0b1000	0b0010	0b101	TLBIP RVALE1IS
0b01	0b000	0b1000	0b0010	0b111	TLBI RVAALE1IS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1000	0b0010	0b111	TLBIP RVAALE1IS
0b01	0b000	0b1000	0b0011	0b000	TLBI VMALLE1IS
0b01	0b000	0b1000	0b0011	0b001	TLBI VAE1IS
0b01	0b000	0b1000	0b0011	0b001	TLBIP VAE1IS
0b01	0b000	0b1000	0b0011	0b010	TLBI ASIDE1IS
0b01	0b000	0b1000	0b0011	0b011	TLBI VAAE1IS
0b01	0b000	0b1000	0b0011	0b011	TLBIP VAAE1IS
0b01	0b000	0b1000	0b0011	0b101	TLBI VALE1IS
0b01	0b000	0b1000	0b0011	0b101	TLBIP VALE1IS
0b01	0b000	0b1000	0b0011	0b111	TLBI VAALE1IS
0b01	0b000	0b1000	0b0011	0b111	TLBIP VAALE1IS
0b01	0b000	0b1000	0b0101	0b001	TLBI RVAE1OS
0b01	0b000	0b1000	0b0101	0b001	TLBIP RVAE1OS
0b01	0b000	0b1000	0b0101	0b011	TLBI RVAAE1OS
0b01	0b000	0b1000	0b0101	0b011	TLBIP RVAAE1OS
0b01	0b000	0b1000	0b0101	0b101	TLBI RVALE1OS
0b01	0b000	0b1000	0b0101	0b101	TLBIP RVALE1OS
0b01	0b000	0b1000	0b0101	0b111	TLBI RVAALE1OS
0b01	0b000	0b1000	0b0101	0b111	TLBIP RVAALE1OS
0b01	0b000	0b1000	0b0110	0b001	TLBI RVAE1
0b01	0b000	0b1000	0b0110	0b001	TLBIP RVAE1
0b01	0b000	0b1000	0b0110	0b011	TLBI RVAAE1
0b01	0b000	0b1000	0b0110	0b011	TLBIP RVAAE1
0b01	0b000	0b1000	0b0110	0b101	TLBI RVALE1
0b01	0b000	0b1000	0b0110	0b101	TLBIP RVALE1
0b01	0b000	0b1000	0b0110	0b111	TLBI RVAALE1
0b01	0b000	0b1000	0b0110	0b111	TLBIP RVAALE1
0b01	0b000	0b1000	0b0111	0b000	TLBI VMALLE1
0b01	0b000	0b1000	0b0111	0b001	TLBI VAE1
0b01	0b000	0b1000	0b0111	0b001	TLBIP VAE1
0b01	0b000	0b1000	0b0111	0b010	TLBI ASIDE1
0b01	0b000	0b1000	0b0111	0b011	TLBI VAAE1
0b01	0b000	0b1000	0b0111	0b011	TLBIP VAAE1
0b01	0b000	0b1000	0b0111	0b101	TLBI VALE1
0b01	0b000	0b1000	0b0111	0b101	TLBIP VALE1
0b01	0b000	0b1000	0b0111	0b111	TLBI VAALE1
0b01	0b000	0b1000	0b0111	0b111	TLBIP VAALE1
0b01	0b000	0b1001	0b0001	0b000	TLBI VMALLE1OSNXS
0b01	0b000	0b1001	0b0001	0b001	TLBI VAE1OSNXS
0b01	0b000	0b1001	0b0001	0b001	TLBIP VAE1OSNXS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1001	0b0001	0b010	TLBI ASIDE1OSNXS
0b01	0b000	0b1001	0b0001	0b011	TLBI VAAE1OSNXS
0b01	0b000	0b1001	0b0001	0b011	TLBIP VAAE1OSNXS
0b01	0b000	0b1001	0b0001	0b101	TLBI VALE1OSNXS
0b01	0b000	0b1001	0b0001	0b101	TLBIP VALE1OSNXS
0b01	0b000	0b1001	0b0001	0b111	TLBI VAALE1OSNXS
0b01	0b000	0b1001	0b0001	0b111	TLBIP VAALE1OSNXS
0b01	0b000	0b1001	0b0010	0b001	TLBI RVAE1ISNXS
0b01	0b000	0b1001	0b0010	0b001	TLBIP RVAE1ISNXS
0b01	0b000	0b1001	0b0010	0b011	TLBI RVAAE1ISNXS
0b01	0b000	0b1001	0b0010	0b011	TLBIP RVAAE1ISNXS
0b01	0b000	0b1001	0b0010	0b101	TLBI RVALE1ISNXS
0b01	0b000	0b1001	0b0010	0b101	TLBIP RVALE1ISNXS
0b01	0b000	0b1001	0b0010	0b111	TLBI RVAALE1ISNXS
0b01	0b000	0b1001	0b0010	0b111	TLBIP RVAALE1ISNXS
0b01	0b000	0b1001	0b0011	0b000	TLBI VMALLE1ISNXS
0b01	0b000	0b1001	0b0011	0b001	TLBI VAE1ISNXS
0b01	0b000	0b1001	0b0011	0b001	TLBIP VAE1ISNXS
0b01	0b000	0b1001	0b0011	0b010	TLBI ASIDE1ISNXS
0b01	0b000	0b1001	0b0011	0b011	TLBI VAAE1ISNXS
0b01	0b000	0b1001	0b0011	0b011	TLBIP VAAE1ISNXS
0b01	0b000	0b1001	0b0011	0b101	TLBI VALE1ISNXS
0b01	0b000	0b1001	0b0011	0b101	TLBIP VALE1ISNXS
0b01	0b000	0b1001	0b0011	0b111	TLBI VAALE1ISNXS
0b01	0b000	0b1001	0b0011	0b111	TLBIP VAALE1ISNXS
0b01	0b000	0b1001	0b0101	0b001	TLBI RVAE1OSNXS
0b01	0b000	0b1001	0b0101	0b001	TLBIP RVAE1OSNXS
0b01	0b000	0b1001	0b0101	0b011	TLBI RVAAE1OSNXS
0b01	0b000	0b1001	0b0101	0b011	TLBIP RVAAE1OSNXS
0b01	0b000	0b1001	0b0101	0b101	TLBI RVALE1OSNXS
0b01	0b000	0b1001	0b0101	0b101	TLBIP RVALE1OSNXS
0b01	0b000	0b1001	0b0101	0b111	TLBI RVAALE1OSNXS
0b01	0b000	0b1001	0b0101	0b111	TLBIP RVAALE1OSNXS
0b01	0b000	0b1001	0b0110	0b001	TLBI RVAE1NXS
0b01	0b000	0b1001	0b0110	0b001	TLBIP RVAE1NXS
0b01	0b000	0b1001	0b0110	0b011	TLBI RVAAE1NXS
0b01	0b000	0b1001	0b0110	0b011	TLBIP RVAAE1NXS
0b01	0b000	0b1001	0b0110	0b101	TLBI RVALE1NXS
0b01	0b000	0b1001	0b0110	0b101	TLBIP RVALE1NXS
0b01	0b000	0b1001	0b0110	0b111	TLBI RVAALE1NXS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1001	0b0110	0b111	TLBIP RVAALE1NXS
0b01	0b000	0b1001	0b0111	0b000	TLBI VMALLE1NXS
0b01	0b000	0b1001	0b0111	0b001	TLBI VAE1NXS
0b01	0b000	0b1001	0b0111	0b001	TLBIP VAE1NXS
0b01	0b000	0b1001	0b0111	0b010	TLBI ASIDE1NXS
0b01	0b000	0b1001	0b0111	0b011	TLBI VAAE1NXS
0b01	0b000	0b1001	0b0111	0b011	TLBIP VAAE1NXS
0b01	0b000	0b1001	0b0111	0b101	TLBI VALE1NXS
0b01	0b000	0b1001	0b0111	0b101	TLBIP VALE1NXS
0b01	0b000	0b1001	0b0111	0b111	TLBI VAALE1NXS
0b01	0b000	0b1001	0b0111	0b111	TLBIP VAALE1NXS
0b01	0b100	0b1000	0b0000	0b001	TLBI IPAS2E1IS
0b01	0b100	0b1000	0b0000	0b001	TLBIP IPAS2E1IS
0b01	0b100	0b1000	0b0000	0b010	TLBI RIPAS2E1IS
0b01	0b100	0b1000	0b0000	0b010	TLBIP RIPAS2E1IS
0b01	0b100	0b1000	0b0000	0b101	TLBI IPAS2LE1IS
0b01	0b100	0b1000	0b0000	0b101	TLBIP IPAS2LE1IS
0b01	0b100	0b1000	0b0000	0b110	TLBI RIPAS2LE1IS
0b01	0b100	0b1000	0b0000	0b110	TLBIP RIPAS2LE1IS
0b01	0b100	0b1000	0b0001	0b000	TLBI ALLE2OS
0b01	0b100	0b1000	0b0001	0b001	TLBI VAE2OS
0b01	0b100	0b1000	0b0001	0b001	TLBIP VAE2OS
0b01	0b100	0b1000	0b0001	0b100	TLBI ALLE1OS
0b01	0b100	0b1000	0b0001	0b101	TLBI VALE2OS
0b01	0b100	0b1000	0b0001	0b101	TLBIP VALE2OS
0b01	0b100	0b1000	0b0001	0b110	TLBI VMALLS12E1OS
0b01	0b100	0b1000	0b0010	0b001	TLBI RVAE2IS
0b01	0b100	0b1000	0b0010	0b001	TLBIP RVAE2IS
0b01	0b100	0b1000	0b0010	0b010	TLBI VMALLWS2E1IS
0b01	0b100	0b1000	0b0010	0b101	TLBI RVALE2IS
0b01	0b100	0b1000	0b0010	0b101	TLBIP RVALE2IS
0b01	0b100	0b1000	0b0011	0b000	TLBI ALLE2IS
0b01	0b100	0b1000	0b0011	0b001	TLBI VAE2IS
0b01	0b100	0b1000	0b0011	0b001	TLBIP VAE2IS
0b01	0b100	0b1000	0b0011	0b100	TLBI ALLE1IS
0b01	0b100	0b1000	0b0011	0b101	TLBI VALE2IS
0b01	0b100	0b1000	0b0011	0b101	TLBIP VALE2IS
0b01	0b100	0b1000	0b0011	0b110	TLBI VMALLS12E1IS
0b01	0b100	0b1000	0b0100	0b000	TLBI IPAS2E1OS
0b01	0b100	0b1000	0b0100	0b000	TLBIP IPAS2E1OS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b100	0b1000	0b0100	0b001	TLBI IPAS2E1
0b01	0b100	0b1000	0b0100	0b001	TLBIP IPAS2E1
0b01	0b100	0b1000	0b0100	0b010	TLBI RIPAS2E1
0b01	0b100	0b1000	0b0100	0b010	TLBIP RIPAS2E1
0b01	0b100	0b1000	0b0100	0b011	TLBI RIPAS2E1OS
0b01	0b100	0b1000	0b0100	0b011	TLBIP RIPAS2E1OS
0b01	0b100	0b1000	0b0100	0b100	TLBI IPAS2LE1OS
0b01	0b100	0b1000	0b0100	0b100	TLBIP IPAS2LE1OS
0b01	0b100	0b1000	0b0100	0b101	TLBI IPAS2LE1
0b01	0b100	0b1000	0b0100	0b101	TLBIP IPAS2LE1
0b01	0b100	0b1000	0b0100	0b110	TLBI RIPAS2LE1
0b01	0b100	0b1000	0b0100	0b110	TLBIP RIPAS2LE1
0b01	0b100	0b1000	0b0100	0b111	TLBI RIPAS2LE1OS
0b01	0b100	0b1000	0b0100	0b111	TLBIP RIPAS2LE1OS
0b01	0b100	0b1000	0b0101	0b001	TLBI RVAE2OS
0b01	0b100	0b1000	0b0101	0b001	TLBIP RVAE2OS
0b01	0b100	0b1000	0b0101	0b010	TLBI VMALLWS2E1OS
0b01	0b100	0b1000	0b0101	0b101	TLBI RVALE2OS
0b01	0b100	0b1000	0b0101	0b101	TLBIP RVALE2OS
0b01	0b100	0b1000	0b0110	0b001	TLBI RVAE2
0b01	0b100	0b1000	0b0110	0b001	TLBIP RVAE2
0b01	0b100	0b1000	0b0110	0b010	TLBI VMALLWS2E1
0b01	0b100	0b1000	0b0110	0b101	TLBI RVALE2
0b01	0b100	0b1000	0b0110	0b101	TLBIP RVALE2
0b01	0b100	0b1000	0b0111	0b000	TLBI ALLE2
0b01	0b100	0b1000	0b0111	0b001	TLBI VAE2
0b01	0b100	0b1000	0b0111	0b001	TLBIP VAE2
0b01	0b100	0b1000	0b0111	0b100	TLBI ALLE1
0b01	0b100	0b1000	0b0111	0b101	TLBI VALE2
0b01	0b100	0b1000	0b0111	0b101	TLBIP VALE2
0b01	0b100	0b1000	0b0111	0b110	TLBI VMALLS12E1
0b01	0b100	0b1001	0b0000	0b001	TLBI IPAS2E1ISNXS
0b01	0b100	0b1001	0b0000	0b001	TLBIP IPAS2E1ISNXS
0b01	0b100	0b1001	0b0000	0b010	TLBI RIPAS2E1ISNXS
0b01	0b100	0b1001	0b0000	0b010	TLBIP RIPAS2E1ISNXS
0b01	0b100	0b1001	0b0000	0b101	TLBI IPAS2LE1ISNXS
0b01	0b100	0b1001	0b0000	0b101	TLBIP IPAS2LE1ISNXS
0b01	0b100	0b1001	0b0000	0b110	TLBI RIPAS2LE1ISNXS
0b01	0b100	0b1001	0b0000	0b110	TLBIP RIPAS2LE1ISNXS
0b01	0b100	0b1001	0b0001	0b000	TLBI ALLE2OSNXS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b100	0b1001	0b0001	0b001	TLBI VAE2OSNXS
0b01	0b100	0b1001	0b0001	0b001	TLBIP VAE2OSNXS
0b01	0b100	0b1001	0b0001	0b100	TLBI ALLE1OSNXS
0b01	0b100	0b1001	0b0001	0b101	TLBI VALE2OSNXS
0b01	0b100	0b1001	0b0001	0b101	TLBIP VALE2OSNXS
0b01	0b100	0b1001	0b0001	0b110	TLBI VMALLS12E1OSNXS
0b01	0b100	0b1001	0b0010	0b001	TLBI RVAE2ISNXS
0b01	0b100	0b1001	0b0010	0b001	TLBIP RVAE2ISNXS
0b01	0b100	0b1001	0b0010	0b010	TLBI VMALLWS2E1ISNXS
0b01	0b100	0b1001	0b0010	0b101	TLBI RVALE2ISNXS
0b01	0b100	0b1001	0b0010	0b101	TLBIP RVALE2ISNXS
0b01	0b100	0b1001	0b0011	0b000	TLBI ALLE2ISNXS
0b01	0b100	0b1001	0b0011	0b001	TLBI VAE2ISNXS
0b01	0b100	0b1001	0b0011	0b001	TLBIP VAE2ISNXS
0b01	0b100	0b1001	0b0011	0b100	TLBI ALLE1ISNXS
0b01	0b100	0b1001	0b0011	0b101	TLBI VALE2ISNXS
0b01	0b100	0b1001	0b0011	0b101	TLBIP VALE2ISNXS
0b01	0b100	0b1001	0b0011	0b110	TLBI VMALLS12E1ISNXS
0b01	0b100	0b1001	0b0100	0b000	TLBI IPAS2E1OSNXS
0b01	0b100	0b1001	0b0100	0b000	TLBIP IPAS2E1OSNXS
0b01	0b100	0b1001	0b0100	0b001	TLBI IPAS2E1NXS
0b01	0b100	0b1001	0b0100	0b001	TLBIP IPAS2E1NXS
0b01	0b100	0b1001	0b0100	0b010	TLBI RIPAS2E1NXS
0b01	0b100	0b1001	0b0100	0b010	TLBIP RIPAS2E1NXS
0b01	0b100	0b1001	0b0100	0b011	TLBI RIPAS2E1OSNXS
0b01	0b100	0b1001	0b0100	0b011	TLBIP RIPAS2E1OSNXS
0b01	0b100	0b1001	0b0100	0b100	TLBI IPAS2LE1OSNXS
0b01	0b100	0b1001	0b0100	0b100	TLBIP IPAS2LE1OSNXS
0b01	0b100	0b1001	0b0100	0b101	TLBI IPAS2LE1NXS
0b01	0b100	0b1001	0b0100	0b101	TLBIP IPAS2LE1NXS
0b01	0b100	0b1001	0b0100	0b110	TLBI RIPAS2LE1NXS
0b01	0b100	0b1001	0b0100	0b110	TLBIP RIPAS2LE1NXS
0b01	0b100	0b1001	0b0100	0b111	TLBI RIPAS2LE1OSNXS
0b01	0b100	0b1001	0b0100	0b111	TLBIP RIPAS2LE1OSNXS
0b01	0b100	0b1001	0b0101	0b001	TLBI RVAE2OSNXS
0b01	0b100	0b1001	0b0101	0b001	TLBIP RVAE2OSNXS
0b01	0b100	0b1001	0b0101	0b010	TLBI VMALLWS2E1OSNXS
0b01	0b100	0b1001	0b0101	0b101	TLBI RVALE2OSNXS
0b01	0b100	0b1001	0b0101	0b101	TLBIP RVALE2OSNXS
0b01	0b100	0b1001	0b0110	0b001	TLBI RVAE2NXS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b100	0b1001	0b0110	0b001	TLBIP RVAE2NXS
0b01	0b100	0b1001	0b0110	0b010	TLBI VMALLWS2E1NXS
0b01	0b100	0b1001	0b0110	0b101	TLBI RVALE2NXS
0b01	0b100	0b1001	0b0110	0b101	TLBIP RVALE2NXS
0b01	0b100	0b1001	0b0111	0b000	TLBI ALLE2NXS
0b01	0b100	0b1001	0b0111	0b001	TLBI VAE2NXS
0b01	0b100	0b1001	0b0111	0b001	TLBIP VAE2NXS
0b01	0b100	0b1001	0b0111	0b100	TLBI ALLE1NXS
0b01	0b100	0b1001	0b0111	0b101	TLBI VALE2NXS
0b01	0b100	0b1001	0b0111	0b101	TLBIP VALE2NXS
0b01	0b100	0b1001	0b0111	0b110	TLBI VMALLS12E1NXS
0b01	0b110	0b1000	0b0001	0b000	TLBI ALLE3OS
0b01	0b110	0b1000	0b0001	0b001	TLBI VAE3OS
0b01	0b110	0b1000	0b0001	0b001	TLBIP VAE3OS
0b01	0b110	0b1000	0b0001	0b100	TLBI PAALLOS
0b01	0b110	0b1000	0b0001	0b101	TLBI VALE3OS
0b01	0b110	0b1000	0b0001	0b101	TLBIP VALE3OS
0b01	0b110	0b1000	0b0010	0b001	TLBI RVAE3IS
0b01	0b110	0b1000	0b0010	0b001	TLBIP RVAE3IS
0b01	0b110	0b1000	0b0010	0b101	TLBI RVALE3IS
0b01	0b110	0b1000	0b0010	0b101	TLBIP RVALE3IS
0b01	0b110	0b1000	0b0011	0b000	TLBI ALLE3IS
0b01	0b110	0b1000	0b0011	0b001	TLBI VAE3IS
0b01	0b110	0b1000	0b0011	0b001	TLBIP VAE3IS
0b01	0b110	0b1000	0b0011	0b101	TLBI VALE3IS
0b01	0b110	0b1000	0b0011	0b101	TLBIP VALE3IS
0b01	0b110	0b1000	0b0100	0b011	TLBI RPAOS
0b01	0b110	0b1000	0b0100	0b111	TLBI RPALOS
0b01	0b110	0b1000	0b0101	0b001	TLBI RVAE3OS
0b01	0b110	0b1000	0b0101	0b001	TLBIP RVAE3OS
0b01	0b110	0b1000	0b0101	0b101	TLBI RVALE3OS
0b01	0b110	0b1000	0b0101	0b101	TLBIP RVALE3OS
0b01	0b110	0b1000	0b0110	0b001	TLBI RVAE3
0b01	0b110	0b1000	0b0110	0b001	TLBIP RVAE3
0b01	0b110	0b1000	0b0110	0b101	TLBI RVALE3
0b01	0b110	0b1000	0b0110	0b101	TLBIP RVALE3
0b01	0b110	0b1000	0b0111	0b000	TLBI ALLE3
0b01	0b110	0b1000	0b0111	0b001	TLBI VAE3
0b01	0b110	0b1000	0b0111	0b001	TLBIP VAE3
0b01	0b110	0b1000	0b0111	0b100	TLBI PAALL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b110	0b1000	0b0111	0b101	TLBI VALE3
0b01	0b110	0b1000	0b0111	0b101	TLBIP VALE3
0b01	0b110	0b1001	0b0001	0b000	TLBI ALLE3OSNXS
0b01	0b110	0b1001	0b0001	0b001	TLBI VAE3OSNXS
0b01	0b110	0b1001	0b0001	0b001	TLBIP VAE3OSNXS
0b01	0b110	0b1001	0b0001	0b101	TLBI VALE3OSNXS
0b01	0b110	0b1001	0b0001	0b101	TLBIP VALE3OSNXS
0b01	0b110	0b1001	0b0010	0b001	TLBI RVAE3ISNXS
0b01	0b110	0b1001	0b0010	0b001	TLBIP RVAE3ISNXS
0b01	0b110	0b1001	0b0010	0b101	TLBI RVALE3ISNXS
0b01	0b110	0b1001	0b0010	0b101	TLBIP RVALE3ISNXS
0b01	0b110	0b1001	0b0011	0b000	TLBI ALLE3ISNXS
0b01	0b110	0b1001	0b0011	0b001	TLBI VAE3ISNXS
0b01	0b110	0b1001	0b0011	0b001	TLBIP VAE3ISNXS
0b01	0b110	0b1001	0b0011	0b101	TLBI VALE3ISNXS
0b01	0b110	0b1001	0b0011	0b101	TLBIP VALE3ISNXS
0b01	0b110	0b1001	0b0101	0b001	TLBI RVAE3OSNXS
0b01	0b110	0b1001	0b0101	0b001	TLBIP RVAE3OSNXS
0b01	0b110	0b1001	0b0101	0b101	TLBI RVALE3OSNXS
0b01	0b110	0b1001	0b0101	0b101	TLBIP RVALE3OSNXS
0b01	0b110	0b1001	0b0110	0b001	TLBI RVAE3NXS
0b01	0b110	0b1001	0b0110	0b001	TLBIP RVAE3NXS
0b01	0b110	0b1001	0b0110	0b101	TLBI RVALE3NXS
0b01	0b110	0b1001	0b0110	0b101	TLBIP RVALE3NXS
0b01	0b110	0b1001	0b0111	0b000	TLBI ALLE3NXS
0b01	0b110	0b1001	0b0111	0b001	TLBI VAE3NXS
0b01	0b110	0b1001	0b0111	0b001	TLBIP VAE3NXS
0b01	0b110	0b1001	0b0111	0b101	TLBI VALE3NXS
0b01	0b110	0b1001	0b0111	0b101	TLBIP VALE3NXS

For more information about these instructions, see TLB maintenance instructions.

C5.1.4.5 Branch Record Buffer instructions

Table C5-5 lists the Branch Record Buffer instructions and their encodings.

Table C5-5 Branch Record Buffer instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b001	0b0111	0b0010	0b100	BRB IALL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b01	0b001	0b0111	0b0010	0b101	BRB INJ

For more information, see The Branch Record Buffer Extension.

C5.1.4.6 Instrumentation Trace Extension instructions

Table C5-6 lists the Instrumentation Trace Extension instructions and their encodings.

Table C5-6 Trace Extension instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b011	0b0111	0b0010	0b111	TRCIT

For more information, see Instrumentation extension.

C5.1.4.7 Guarded Control Stack instructions

Table C5-7 lists the Guarded Control Stack instructions and their encodings.

Table C5-7 Guarded Control Stack instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b0111	0b100	GCSPUSHX
0b01	0b000	0b0111	0b0111	0b101	GCSPOPCX
0b01	0b000	0b0111	0b0111	0b110	GCSPOPX
0b01	0b011	0b0111	0b0111	0b000	GCSPUSHM
0b01	0b011	0b0111	0b0111	0b001	GCSPOPM
0b01	0b011	0b0111	0b0111	0b010	GCSSS1
0b01	0b011	0b0111	0b0111	0b011	GCSSS2

For more information, see The Guarded Control Stack.

C5.1.4.8 Reserved encoding space for IMPLEMENTATION DEFINED instructions

The A64 instruction set reserves the following encoding space for IMPLEMENTATION DEFINED instructions:

The value of `L` defines the use of `Rt` as follows:

0 `Rt` is an argument supplied to the instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1 `Rt` is a result returned by the instruction.

IMPLEMENTATION DEFINED instructions in this encoding space are accessed using the `SYS`, `SYSL`, and `SYSP` instructions, see SYS, SYSL and SYSP.

See also Reserved encodings for IMPLEMENTATION DEFINED registers.

C5.1.5 `op0`== `0b11`, Moves to and from Special-purpose registers

The instructions that move data to and from non-debug System registers are encoded with `op0`== `0b11`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15 … 12	11 … 8	7 … 5	4 … 0
1	1	0	1	0	1	0	1	0	0	L	1	1	op1	CRn	CRm	op2	Rt
											op0						

C5.1.5.1 Instructions for accessing Special-purpose registers

The value of `CRn` provides the next level of decode of these instructions. For Special-purpose registers, the value of CRn is 4.

The A64 instructions for accessing Special-purpose registers are:

```
MSR <Special-purpose_register>, Xt    ; Write to Special-purpose register
MRS Xt, <Special-purpose_register>    ; Read from Special-purpose register
```

Where `<Special-purpose_register>` is the register name, for example SP_EL0.

For these accesses, `CRn` has the value `0b0100`. The encoding for Special-purpose register accesses is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15	14	13	12	11 … 8	7 … 5	4 … 0
1	1	0	1	0	1	0	1	0	0	L	1	1	op1	0	1	0	0	CRm	op2	Rt
											op0			CRn						

The full list of Special-purpose registers is in Table C5-8. The characteristic of a Special-purpose register is that all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

Table C5-8 lists the encodings for `op1`, `CRm`, and `op2` fields for accesses to the Special-purpose registers in AArch64.

Table C5-8 Instruction encodings for Special-purpose register accesses

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
`0b000`	`0b0100`	`0b0000`	`0b000`	RW	SPSR_EL1	SPSR_EL1 SPSR_EL2
`0b000`	`0b0100`	`0b0000`	`0b001`	RW	ELR_EL1	ELR_EL1 ELR_EL2
`0b000`	`0b0100`	`0b0001`	`0b000`	RW	SP_EL0	SP_EL0
`0b000`	`0b0100`	`0b0010`	`0b000`	RW	SPSel	SPSel

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0100	0b0010	0b010	RO	CurrentEL	CurrentEL
0b000	0b0100	0b0010	0b011	RW	PAN	PAN
0b000	0b0100	0b0010	0b100	RW	UAO	UAO
0b000	0b0100	0b0011	0b000	RW	ALLINT	ALLINT
0b000	0b0100	0b0011	0b001	RW	PM	PM
0b000	0b0100	0b0110	0b000	RW	ICC_PMR_EL1	ICC_PMR_EL1 ICV_PMR_EL1
0b011	0b0100	0b0010	0b000	RW	NZCV	NZCV
0b011	0b0100	0b0010	0b001	RW	DAIF	DAIF
0b011	0b0100	0b0010	0b010	RW	SVCR	SVCR
0b011	0b0100	0b0010	0b101	RW	DIT	DIT
0b011	0b0100	0b0010	0b110	RW	SSBS	SSBS
0b011	0b0100	0b0010	0b111	RW	TCO	TCO
0b011	0b0100	0b0100	0b000	RW	FPCR	FPCR
0b011	0b0100	0b0100	0b001	RW	FPSR	FPSR
0b011	0b0100	0b0100	0b010	RW	FPMR	FPMR
0b011	0b0100	0b0101	0b000	RW	DSPSR_EL0	DSPSR_EL0
0b011	0b0100	0b0101	0b001	RW	DLR_EL0	DLR_EL0
0b100	0b0100	0b0000	0b000	RW	SPSR_EL2	SPSR_EL1 SPSR_EL2
0b100	0b0100	0b0000	0b001	RW	ELR_EL2	ELR_EL1 ELR_EL2
0b100	0b0100	0b0001	0b000	RW	SP_EL1	SP_EL1
0b100	0b0100	0b0011	0b000	RW	SPSR_irq	SPSR_irq
0b100	0b0100	0b0011	0b001	RW	SPSR_abt	SPSR_abt
0b100	0b0100	0b0011	0b010	RW	SPSR_und	SPSR_und
0b100	0b0100	0b0011	0b011	RW	SPSR_fiq	SPSR_fiq
0b101	0b0100	0b0000	0b000	RW	SPSR_EL12	SPSR_EL1
0b101	0b0100	0b0000	0b001	RW	ELR_EL12	ELR_EL1
0b110	0b0100	0b0000	0b000	RW	SPSR_EL3	SPSR_EL3
0b110	0b0100	0b0000	0b001	RW	ELR_EL3	ELR_EL3
0b110	0b0100	0b0001	0b000	RW	SP_EL2	SP_EL2

All direct and indirect reads and writes to Special-purpose registers appear to occur in program order relative to other instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2 Special-purpose registers

This section describes the following Special-purpose registers:

- ALLINT, that holds the PSTATE.ALLINT bit.
- CurrentEL, that holds PSTATE.EL, and that software can read to determine the current Exception level.
- DAIF, that holds the current PSTATE.{D, A, I, F} interrupt mask bits.
- DIT, that holds the PSTATE.DIT bit.
- ELR_EL1, that holds the address to return to for an exception return from EL1.
- ELR_EL2, that holds the address to return to for an exception return from EL2.
- ELR_EL3, that holds the address to return to for an exception return from EL3.
- FPCR, that provides control of floating-point operation.
- FPMR, that controls behaviors of the FP8 instructions.
- FPSR, that provides floating-point status information.
- NZCV, that holds the PSTATE.{N, Z, C, V} condition flags.
- PAN, that holds the PSTATE.PAN state bit.
- PM, that provides access to the Profiling exception mask bit.
- SP_EL0, that holds the stack pointer for EL0.
- SP_EL1, that holds the stack pointer for EL1.
- SP_EL2, that holds the stack pointer for EL2.
- SP_EL3, that holds the stack pointer for EL3.
- SPSel, that holds PSTATE.SP, that at EL1 or higher selects the current SP.
- SPSR_abt, that holds process state on taking an exception to AArch32 Abort mode.
- SPSR_EL1, that holds process state on taking an exception to AArch64 EL1.
- SPSR_EL2, that holds process state on taking an exception to AArch64 EL2.
- SPSR_EL3, that holds process state on taking an exception to AArch64 EL3.
- SPSR_fiq, that holds process state on taking an exception to AArch32 FIQ mode.
- SPSR_irq, that holds process state on taking an exception to AArch32 IRQ mode.
- SPSR_und, that holds process state on taking an exception to AArch32 Undefined mode.
- SSBS, that holds the PSTATE.SSBS bit.
- SVCR, controls Streaming SVE mode and SME behavior.
- TCO, that holds the PSTATE.TCO bit.
- UAO, that holds the PSTATE.UAO bit.

The following registers are also Special-purpose registers:

- DLR_EL0, that holds the address to return to for a return from Debug state.
- DSPSR_EL0, that holds process state on entry to Debug state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.1 ALLINT, All Interrupt Mask Bit

The ALLINT characteristics are:

Purpose

Allows access to the all interrupt mask bit.

Configuration

This register is present only when FEAT_NMI is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ALLINT are UNDEFINED.

Attributes

ALLINT is a 64-bit register.

Field descriptions

Bits [63:14]

Reserved, RES0.

ALLINT, bit [13]

All interrupt mask. An interrupt is controlled by PSTATE.ALLINT when all of the following apply:

- SCTLR_ELx.NMI is 1.
- The interrupt is targeted at ELx.
- Execution is at ELx.

ALLINT	Meaning
0b0	This control does not cause any interrupts to be masked.
0b1	If SCTLR_ELx.NMI is 1 and execution is at ELx, an IRQ or FIQ interrupt that is targeted to ELx, with or without Superpriority, is masked.

The value of this bit is set to the inverse value in the SCTLR_ELx.SPINTMASK field on taking an exception to ELx.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [12:0]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing ALLINT

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ALLINT
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_NMI) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(50):PSTATE.ALLINT:Zeros(13);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(50):PSTATE.ALLINT:Zeros(13);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(50):PSTATE.ALLINT:Zeros(13);
```

```
MSR ALLINT, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_NMI) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PSTATE.ALLINT = X[t, 64]<13>;
elsif PSTATE.EL == EL2 then
    PSTATE.ALLINT = X[t, 64]<13>;
elsif PSTATE.EL == EL3 then
    PSTATE.ALLINT = X[t, 64]<13>;
```

```
MSR ALLINT, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b001	0b0100	0b000x	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.2 CurrentEL, Current Exception Level

The CurrentEL characteristics are:

Purpose

Holds the current Exception level.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CurrentEL are UNDEFINED.

Attributes

CurrentEL is a 64-bit register.

Field descriptions

Bits [63:4]

Reserved, RES0.

EL, bits [3:2]

Current Exception level.

EL	Meaning
`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

When the Effective value of HCR_EL2.NV is 1, EL1 read accesses to the CurrentEL register return the value of `0b10` in this field.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'01'`.
  - When the highest implemented Exception level is EL2, this field resets to `'10'`.
  - Otherwise, this field resets to `'11'`.

Bits [1:0]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CurrentEL

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CurrentEL
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        X[t, 64] = Zeros(60):'10':Zeros(2);
    else
        X[t, 64] = Zeros(60):PSTATE.EL:Zeros(2);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(60):PSTATE.EL:Zeros(2);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(60):PSTATE.EL:Zeros(2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.3 DAIF, Interrupt Mask Bits

The DAIF characteristics are:

Purpose

Allows access to the interrupt mask bits.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DAIF are UNDEFINED.

Attributes

DAIF is a 64-bit register.

Field descriptions

Bits [63:10]

Reserved, RES0.

D, bit [9]

Process state D mask.

D	Meaning
`0b0`	Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.
`0b1`	Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

A, bit [8]

SError exception mask bit.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

I, bit [7]

IRQ mask bit.

I	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

F, bit [6]

FIQ mask bit.

F	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Bits [5:0]

Reserved, RES0.

Accessing DAIF

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DAIF
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if ELIsInHost(EL0) || SCTLR_EL1.UMA == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
```

MSR DAIF, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if ELIsInHost(EL0) || SCTLR_EL1.UMA == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
elsif PSTATE.EL == EL1 then
    PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
elsif PSTATE.EL == EL2 then
    PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
elsif PSTATE.EL == EL3 then
    PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
```

MSR DAIFSet, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b110

MSR DAIFClr, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b111

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.4 DIT, Data Independent Timing

The DIT characteristics are:

Purpose

Allows access to the Data Independent Timing bit.

Configuration

This register is present only when FEAT_DIT is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DIT are UNDEFINED.

Attributes

DIT is a 64-bit register.

Field descriptions

Bits [63:25]

Reserved, RES0.

DIT, bit [24]

Data Independent Timing.

DIT	Meaning
0b0	The architecture makes no statement about the timing properties of any instructions.
0b1	The architecture requires that the execution time of a data-independent-time sequence of code must be independent of all data-independent-time values. For more information, see About PSTATE.DIT.

The list of data-independent-time instructions is:

- Flag Manipulation: CFINV.
- Data Processing – Immediate:
  - Add/subtract (immediate): ADD, ADDS, SUB, and SUBS.
  - Bitfield: BFM, SBFM, and UBFM.
  - Extract: EXTR.
  - Logical (immediate): AND, ANDS, EOR, and ORR.
  - Min/max (immediate): SMAX, SMIN, UMAX, and UMIN.
- Data Processing – Register:
  - Add/subtract (extended register): ADD, ADDS, SUB, and SUBS.
  - Add/subtract (shifted register): ADD, ADDS, SUB, and SUBS.
  - Add/subtract (with carry): ADC, ADCS, SBC, and SBCS.
  - Conditional compare (immediate): CCMN, and CCMP.
  - Conditional compare (register): CCMN, and CCMP.
  - Conditional select: CSEL, CSINC, CSINV, and CSNEG.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Data-processing (1 source): ABS, CLS, CLZ, CNT, CTZ, RBIT, REV16, REV32, and REV.
  - Data-processing (2 source): ASRV, CRC32B, CRC32CB, CRC32CH, CRC32CW, CRC32CX, CRC32H, CRC32W, CRC32X, LSLV, LSRV, RORV, SMAX, SMIN, UMAX, and UMIN.
  - Data-processing (3 source): MADD, MSUB, SMADDL, SMSUBL, SMULH, UMADDL, UMSUBL, and UMULH.
  - Evaluate into flags: SETF16, and SETF8.
  - Logical (shifted register): AND, ANDS, BIC, BICS, EON, EOR, ORN, and ORR.
  - Rotate right into flags: RMIF.
- Data Processing – Scalar Floating-Point and Advanced SIMD:
  - Advanced SIMD across lanes: ADDV, SADDLV, SMAXV, SMINV, UADDLV, UMAXV, and UMINV.
  - Advanced SIMD copy: DUP, INS, SMOV, and UMOV.
  - Advanced SIMD extract: EXT.
  - Advanced SIMD modified immediate: BIC, and ORR.
  - Advanced SIMD permute: TRN1, TRN2, UZP1, UZP2, ZIP1, and ZIP2.
  - Advanced SIMD scalar copy: DUP.
  - Advanced SIMD scalar pairwise: ADDP.
  - Advanced SIMD scalar shift by immediate: Advanced SIMD scalar shift by immediate: SHL, SLI, SRSHR, SRI, SRSRA, SSHR, SSRA, URSHR, URSRA, USHR, and USRA.
  - Advanced SIMD scalar three same: ADD, CMEQ, CMGE, CMGT, CMHI, CMHS, CMTST, SQDMULH, SQRDMULH, SSHL, SUB, and USHL.
  - Advanced SIMD scalar three same extra: SQRDMLAH.
  - Advanced SIMD scalar two-register miscellaneous: ABS, CMEQ, CMGE, CMGT, CMLE, CMLT, and NEG.
  - Advanced SIMD scalar x indexed element: SQDMULH, SQRDMLAH, and SQRDMULH.
  - Advanced SIMD shift by immediate: RSHRN2, RSHRN, SHL, SHRN2, SHRN, SLI, SRI, SSHLL2, SSHLL, SSHR, SSRA, USHLL2, USHLL, USHR, and USRA.
  - Advanced SIMD table lookup: LUTI2, LUTI4, TBL, and TBX.
  - Advanced SIMD three different: ADDHN2, ADDHN, PMULL2, PMULL, RADDHN2, RADDHN, RSUBHN2, RSUBHN, SABAL2, SABAL, SABDL2, SABDL, SADDL2, SADDL, SADDW2, SADDW, SMLAL2, SMLAL, SMLSL2, SMLSL, SMULL2, SMULL, SSUBL2, SSUBL, SSUBW2, SSUBW, SUBHN2, SUBHN, UABAL2, UABAL, UABDL2, UABDL, UADDL2, UADDL, UADDW2, UADDW, UMLAL2, UMLAL, UMLSL2, UMLSL, UMULL2, UMULL, USUBL2, USUBL, USUBW2, and USUBW.
  - Advanced SIMD three same: ADD, ADDP, AND, BIC, BIF, BIT, BSL, CMEQ, CMGE, CMGT, CMHI, CMHS, CMTST, EOR, MLA, MLS, MUL, ORN, ORR, PMUL, SABA, SABD, SHADD, SHSUB, SMAX, SMAXP, SMIN, SMINP, SQDMULH, SQRDMULH, SSHL, SUB, UABA, UABD, UHADD, UHSUB, UMAX, UMAXP, UMIN, UMINP, and USHL.
  - Advanced SIMD three-register extension: SDOT, SMMLA, SQRDMLAH, UDOT, UMMLA, USDOT, and USMMLA.
  - Advanced SIMD two-register miscellaneous: ABS, CLS, CLZ, CMEQ, CMGE, CMGT, CMLE, CMLT, CNT, NEG, NOT, RBIT, REV16, REV32, REV64, SADALP, SADDLP, SHLL2, SHLL, UADALP, UADDLP, XTN2, and XTN.
  - Advanced SIMD vector x indexed element: MLA, MLS, MUL, SDOT, SMLAL2, SMLAL, SMLSL2, SMLSL, SMULL2, SMULL, SQDMULH, SQRDMLAH, SQRDMULH, SUDOT, UDOT, UMLAL2, UMLAL, UMLSL2, UMLSL, UMULL2, UMULL, and USDOT.
  - Cryptographic AES: AESD, AESE, AESIMC, and AESMC.
  - Cryptographic four-register: BCAX, EOR3, and SM3SS1.
  - Cryptographic three-register SHA: SHA1C, SHA1M, SHA1P, SHA1SU0, SHA256H2, SHA256H, and SHA256SU1.
  - Cryptographic three-register SHA 512: RAX1, SHA512H2, SHA512H, SHA512SU1, SM3PARTW1, SM3PARTW2, and SM4EKEY.
  - Cryptographic three-register, imm2: SM3TT1A, SM3TT1B, SM3TT2A, and SM3TT2B.
  - Cryptographic three-register, imm6: XAR.
  - Cryptographic two-register SHA: SHA1H, SHA1SU1, and SHA256SU0.
  - Cryptographic two-register SHA 512: SHA512SU0, and SM4E.
  - Floating-point conditional select: FCSEL.
- Loads and Stores:
  - 128-bit atomic memory operations: LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL, LDSETP, LDSETPA, LDSETPAL, LDSETPL, SWPP, SWPPA, SWPPAL, and SWPPL.
  - Advanced SIMD load/store multiple structures: LD1, LD2, LD3, LD4, ST1, ST2, ST3, and ST4.
  - Advanced SIMD load/store multiple structures (post-indexed): LD1, LD2, LD3, LD4, ST1, ST2, ST3, and ST4.
  - Advanced SIMD load/store single structure: LD1, LD1R, LD2, LD2R, LD3, LD3R, LD4, LD4R, LDAP1, ST1, ST2, ST3, ST4, and STL1.
  - Advanced SIMD load/store single structure (post-indexed): LD1, LD1R, LD2, LD2R, LD3, LD3R, LD4, LD4R, ST1, ST2, ST3, and ST4.
  - Atomic memory operations: LDADD, LDADDA, LDADDAB, LDADDAH, LDADDAL, LDADDALB, LDADDALH, LDADDB, LDADDH, LDADDL, LDADDLB, LDADDLH, LDAPR, LDAPRB, LDAPRH, LDCLR, LDCLRA, LDCLRAB, LDCLRAH, LDCLRAL, LDCLRALB, LDCLRALH, LDCLRB, LDCLRH, LDCLRL, LDCLRLB, LDCLRLH, LDEOR, LDEORA, LDEORAB, LDEORAH, LDEORAL, LDEORALB, LDEORALH, LDEORB, LDEORH, LDEORL, LDEORLB,

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LDEORLH, LDSET, LDSETA, LDSETAB, LDSETAH, LDSETAL, LDSETALB, LDSETALH, LDSETB, LDSETH, LDSETL, LDSETLB, LDSETLH, LDSMAX, LDSMAXA, LDSMAXAB, LDSMAXAH, LDSMAXAL, LDSMAXALB, LDSMAXALH, LDSMAXB, LDSMAXH, LDSMAXL, LDSMAXLB, LDSMAXLH, LDSMIN, LDSMINA, LDSMINAB, LDSMINAH, LDSMINAL, LDSMINALB, LDSMINALH, LDSMINB, LDSMINH, LDSMINL, LDSMINLB, LDSMINLH, LDUMAX, LDUMAXA, LDUMAXAB, LDUMAXAH, LDUMAXAL, LDUMAXALB, LDUMAXALH, LDUMAXB, LDUMAXH, LDUMAXL, LDUMAXLB, LDUMAXLH, LDUMIN, LDUMINA, LDUMINAB, LDUMINAH, LDUMINAL, LDUMINALB, LDUMINALH, LDUMINB, LDUMINH, LDUMINL, LDUMINLB, and LDUMINLH.

- LDAPR/STLR (SIMD&FP): LDAPUR, and STLUR.
- LDAPR/STLR (unscaled immediate): LDAPUR, LDAPURB, LDAPURH, LDAPURSB, LDAPURSH, LDAPURSW, STLUR, STLURB, and STLURH.
- LDAPR/STLR (writeback): LDAPR, and STLR.
- LDIAPP/STILP: LDIAPP, and STILP.
- Load/store exclusive pair: LDAXP, LDXP, STLXP, and STXP.
- Load/store exclusive register: LDAXR, LDAXRB, LDAXRH, LDXR, LDXRB, LDXRH, STLXR, STLXRB, STLXRH, STXR, STXRB and STXRH.
- Load/store no-allocate pair (offset): LDNP, and STNP.
- Load/store ordered: LDAR, LDARB, LDARH, LDLAR, LDLARB, LDLARH, STLLR, STLLRB, STLLRH, STLR, STLRB, and STLRH.
- Load/store register (immediate post-indexed): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (immediate pre-indexed): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (pac): LDRAA, and LDRAB.
- Load/store register (register offset): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (unprivileged): LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW, STTR, STTRB, and STTRH.
- Load/store register (unscaled immediate): LDUR, LDURB, LDURH, LDURSB, LDURSH, LDURSW, STUR, STURB, and STURH.
- Load/store register (unsigned immediate): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register pair (offset): LDP, LDPSW, and STP.
- Load/store register pair (post-indexed): LDP, LDPSW, and STP.
- Load/store register pair (pre-indexed): LDP, LDPSW, and STP.

- If FEAT_SME is implemented, SME encodings:
  - SME Add Vector to Array: ADDHA, and ADDVA.
  - SME Integer Outer Product - 32 bit: SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS.
  - SME Memory: LD1B, LD1D, LD1H, LD1Q, LD1W, LDR, ST1B, ST1D, ST1H, ST1Q, ST1W, and STR.
  - SME Move from Array: MOVA, and MOVAZ.
  - SME Move into Array: MOVA.
  - SME Outer Product - 64 bit:
    - SME Int16 outer product: SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS.
  - SME zero array: ZERO.
  - SME2 Expand Lookup Table (Contiguous): LUTI2, and LUTI4.
  - SME2 Expand Lookup Table (Non-contiguous): LUTI2, and LUTI4.
  - SME2 Move Lookup Table: MOVT.
  - SME2 Multi-vector - Indexed (Four registers):
    - SME2 multi-vec indexed long MLA four sources: SMLAL, SMLSL, UMLAL, and UMLSL.
    - SME2 multi-vec indexed long long MLA four sources 32-bit: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
    - SME2 multi-vec indexed long long MLA four sources 64-bit: SMLALL, SMLSLL, UMLALL, and UMLSLL.
    - SME2 multi-vec ternary indexed four registers 32-bit: SDOT, SUDOT, SUVDOT, SVDOT, UDOT, USDOT, USVDOT, and UVDOT.
    - SME2 multi-vec ternary indexed four registers 64-bit: SDOT, SVDOT, UDOT, and UVDOT.
  - SME2 Multi-vector - Indexed (One register):
    - SME2 multi-vec indexed long MLA one source: SMLAL, SMLSL, UMLAL, and UMLSL.
    - SME2 multi-vec indexed long long MLA one source 32-bit: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
    - SME2 multi-vec indexed long long MLA one source 64-bit: SMLALL, SMLSLL, UMLALL, and UMLSLL.
  - SME2 Multi-vector - Indexed (Two registers):
    - SME2 multi-vec indexed long MLA two sources: SMLAL, SMLSL, UMLAL, and UMLSL.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SME2 multi-vec indexed long long MLA two sources 32-bit: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
  - SME2 multi-vec indexed long long MLA two sources 64-bit: SMLALL, SMLSLL, UMLALL, and UMLSLL.
  - SME2 multi-vec ternary indexed two registers 32-bit: SDOT, SUDOT, SVDOT, UDOT, USDOT, and UVDOT.
  - SME2 multi-vec ternary indexed two registers 64-bit: SDOT, and UDOT.
- SME2 Multi-vector - Memory (Contiguous): LD1B, LD1D, LD1H, LD1W, LDNT1B, LDNT1D, LDNT1H, LDNT1W, ST1B, ST1D, ST1H, ST1W, STNT1B, STNT1D, STNT1H, and STNT1W.
- SME2 Multi-vector - Memory (Strided): LD1B, LD1D, LD1H, LD1W, LDNT1B, LDNT1D, LDNT1H, LDNT1W, ST1B, ST1D, ST1H, ST1W, STNT1B, STNT1D, STNT1H, and STNT1W.
- SME2 Multi-vector - Multiple Array Vectors (Four registers):
  - SME2 multiple vectors binary int four registers: ADD, and SUB.
  - SME2 multiple vectors four-way dot product four registers: SDOT, and UDOT.
  - SME2 multiple vectors long MLA four sources: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 multiple vectors long long MLA four sources: SMLALL, SMLSLL, UMLALL, UMLSLL, and USMLALL.
  - SME2 multiple vectors mixed dot product four registers: USDOT.
  - SME2 multiple vectors ternary int four registers: ADD, and SUB.
  - SME2 multiple vectors two-way dot product four registers: SDOT, and UDOT.
- SME2 Multi-vector - Multiple Array Vectors (Two registers):
  - SME2 multiple vectors binary int two registers: ADD, and SUB.
  - SME2 multiple vectors four-way dot product two registers: SDOT, and UDOT.
  - SME2 multiple vectors long MLA two sources: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 multiple vectors long long MLA two sources: SMLALL, SMLSLL, UMLALL, UMLSLL, and USMLALL.
  - SME2 multiple vectors mixed dot product two registers: USDOT.
  - SME2 multiple vectors ternary int two registers: ADD, and SUB.
  - SME2 multiple vectors two-way dot product two registers: SDOT, and UDOT.
- SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers):
  - SME2 multiple vectors int min/max four registers: SMAX, SMIN, UMAX, and UMIN.
- SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers):
  - SME2 multiple vectors int min/max two registers: SMAX, SMIN, UMAX, and UMIN.
- SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers): SQDMULH.
- SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers): SQDMULH.
- SME2 Multi-vector - Multiple and Single Array Vectors (Four registers):
  - SME2 single-multi four-way dot product four registers: SDOT, and UDOT.
  - SME2 single-multi long MLA four sources: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 single-multi long long MLA four sources: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
  - SME2 single-multi mixed dot product four registers: SUDOT, and USDOT.
  - SME2 single-multi ternary int four registers: ADD, and SUB.
  - SME2 single-multi two-way dot product four registers: SDOT, and UDOT.
- SME2 Multi-vector - Multiple and Single Array Vectors (Two registers):
  - SME2 multiple and single vector long MLA one source: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 multiple and single vector long long FMA one source: SMLALL, SMLSLL, UMLALL, UMLSLL, and USMLALL.
  - SME2 single-multi four-way dot product two registers: SDOT, and UDOT.
  - SME2 single-multi long MLA two sources: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 single-multi long long MLA two sources: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
  - SME2 single-multi mixed dot product two registers: SUDOT, and USDOT.
  - SME2 single-multi ternary int two registers: ADD, and SUB.
  - SME2 single-multi two-way dot product two registers: SDOT, and UDOT.
- SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers):
  - SME2 single-multi add four registers: ADD.
  - SME2 single-multi int min/max four registers: SMAX, SMIN, UMAX, and UMIN.
  - SME2 single-multi signed saturating doubling multiply high four registers: SQDMULH.
- SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers):
  - SME2 single-multi add two registers: ADD.
  - SME2 single-multi int min/max two registers: SMAX, SMIN, UMAX, and UMIN.
  - SME2 single-multi signed saturating doubling multiply high two registers: SQDMULH.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SME2 Multi-vector - SVE Constructive Binary:
  - SME2 multi-vec CLAMP four registers: `SCLAMP`, and `UCLAMP`.
  - SME2 multi-vec CLAMP two registers: `SCLAMP`, and `UCLAMP`.
  - SME2 multi-vec ZIP two registers: `UZP`, and `ZIP`.
  - SME2 multi-vec quadwords ZIP two registers: `UZP`, and `ZIP`.
- SME2 Multi-vector - SVE Constructive Unary:
  - SME2 multi-vec ZIP four registers: `UZP`, and `ZIP`.
  - SME2 multi-vec quadwords ZIP four registers: `UZP`, and `ZIP`.
  - SME2 multi-vec unpack four registers: `SUNPK`, and `UUNPK`.
  - SME2 multi-vec unpack two registers: `SUNPK`, and `UUNPK`.
- SME2 Multi-vector - SVE Select: `SEL`.
- SME2 Multiple Zero: `ZERO`.
- SME2 Outer Product - Misc:
  - SME2 32-bit binary outer product: `BMOPA`, and `BMOPS`.

- If FEAT_SVE is implemented, SVE encodings:
  - SVE Address Generation: `ADR`.
  - SVE Bitwise Immediate: `AND`, `EOR`, and `ORR`.
  - SVE Bitwise Logical - Unpredicated: `AND`, `BCAX`, `BIC`, `BSL1N`, `BSL2N`, `BSL`, `EOR3`, `EOR`, `NBSL`, `ORR`, and `XAR`.
  - SVE Bitwise Shift - Predicated:
    - SVE bitwise shift by immediate (predicated): `ASR`, `ASRD`, `LSL`, `LSR`, `SRSHR`, and `URSHR`.
    - SVE bitwise shift by vector (predicated): `ASR`, `ASRR`, `LSL`, `LSLR`, `LSR`, and `LSRR`.
    - SVE bitwise shift by wide elements (predicated): `ASR`, `LSL`, and `LSR`.
  - SVE Bitwise Shift - Unpredicated: `ASR`, `LSL`, and `LSR`.
  - SVE Element Count:
    - SVE inc/dec register by element count: `DECB`, `DECD`, `DECH`, `DECW`, `INCB`, `INCD`, `INCH`, and `INCW`.
    - SVE inc/dec vector by element count: `DECD`, `DECH`, `DECW`, `INCD`, `INCH`, and `INCW`.
  - SVE Integer Arithmetic - Unpredicated:
    - SVE integer add/subtract vectors (unpredicated): `ADD`, and `SUB`.
  - SVE Integer Binary Arithmetic - Predicated:
    - SVE bitwise logical operations (predicated): `AND`, `BIC`, `EOR`, and `ORR`.
    - SVE integer add/subtract vectors (predicated): `ADD`, `SUB`, and `SUBR`.
    - SVE integer min/max/difference (predicated): `SABD`, `SMAX`, `SMIN`, `UABD`, `UMAX`, and `UMIN`.
    - SVE integer multiply vectors (predicated): `MUL`, `SMULH`, and `UMULH`.
  - SVE Integer Compare - Scalars: `CTERMEQ`, and `CTERMNE`.
  - SVE Integer Compare - Signed Immediate: `CMP<cc>`.
  - SVE Integer Compare - Unsigned Immediate: `CMP<cc>`.
  - SVE Integer Compare - Vectors: `CMP<cc>`.
  - SVE Integer Misc - Unpredicated: `MOVPRFX`.
  - SVE Integer Multiply-Add - Predicated: `MAD`, `MLA`, `MLS`, and `MSB`.
  - SVE Integer Multiply-Add - Unpredicated:
    - SVE integer dot product (unpredicated): `SDOT`, and `UDOT`.
    - SVE mixed sign dot product: `USDOT`.
    - SVE2 complex integer multiply-add: `CMLA`.
    - SVE2 integer multiply-add long: `SMLALB`, `SMLALT`, `SMLSLB`, `SMLSLT`, `UMLALB`, `UMLALT`, `UMLSLB`, and `UMLSLT`.
    - SVE2 saturating multiply-add high: `SQRDMLAH`.
  - SVE Integer Reduction: `ADDQV`, `ANDQV`, `ANDV`, `EORQV`, `EORV`, `MOVPRFX`, `ORQV`, `ORV`, `SADDV`, `SMAXQV`, `SMAXV`, `SMINQV`, `SMINV`, `UADDV`, `UMAXQV`, `UMAXV`, `UMINQV`, and `UMINV`.
  - SVE Integer Unary Arithmetic - Predicated:
    - SVE bitwise unary operations (predicated): `CLS`, `CLZ`, `CNOT`, `CNT`, and `NOT`.
    - SVE integer unary operations (predicated): `ABS`, `NEG`, `SXTB`, `SXTH`, `SXTW`, `UXTB`, `UXTH`, and `UXTW`.
  - SVE Integer Wide Immediate - Unpredicated:
    - SVE integer add/subtract immediate (unpredicated): `ADD`, `SUB`, and `SUBR`.
    - SVE integer min/max immediate (unpredicated): `SMAX`, `SMIN`, `UMAX`, and `UMIN`.
    - SVE integer multiply immediate (unpredicated): `MUL`.
  - SVE Memory - 32-bit Gather and Unsized Contiguous:
    - SVE 32-bit gather load (scalar plus 32-bit unscaled offsets): `LD1B`, `LD1H`, `LD1SB`, `LD1SH`, and `LD1W`.
    - SVE 32-bit gather load (vector plus immediate): `LD1B`, `LD1H`, `LD1SB`, `LD1SH`, and `LD1W`.
    - SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets): `LD1H`, and `LD1SH`.
    - SVE 32-bit gather load words (scalar plus 32-bit scaled offsets): `LD1W`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SVE load and broadcast element: LD1RB, LD1RD, LD1RH, LD1RSB, LD1RSH, LD1RSW, and LD1RW.
  - SVE load vector register: LDR.
  - SVE2 32-bit gather non-temporal load (vector plus scalar): LDNT1B, LDNT1H, LDNT1SB, LDNT1SH, and LDNT1W.
- SVE Memory - 64-bit Gather:
  - SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets): LD1D, LD1H, LD1SH, LD1SW, and LD1W.
  - SVE 64-bit gather load (scalar plus 64-bit scaled offsets): LD1D, LD1H, LD1SH, LD1SW, and LD1W.
  - SVE 64-bit gather load (scalar plus 64-bit unscaled offsets): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE 64-bit gather load (vector plus immediate): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE2 128-bit gather load (vector plus scalar): LD1Q.
  - SVE2 64-bit gather non-temporal load (vector plus scalar): LDNT1B, LDNT1D, LDNT1H, LDNT1SB, LDNT1SH, LDNT1SW, and LDNT1W.
- SVE Memory - Contiguous Load:
  - SVE contiguous load (quadwords, scalar plus immediate): LD1D, and LD1W.
  - SVE contiguous load (quadwords, scalar plus scalar): LD1D, and LD1W.
  - SVE contiguous load (scalar plus immediate): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE contiguous load (scalar plus scalar): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE contiguous non-temporal load (scalar plus immediate): LDNT1B, LDNT1D, LDNT1H, and LDNT1W.
  - SVE contiguous non-temporal load (scalar plus scalar): LDNT1B, LDNT1D, LDNT1H, and LDNT1W.
  - SVE load and broadcast quadword (scalar plus immediate): LD1ROB, LD1ROD, LD1ROH, LD1ROW, LD1RQB, LD1RQD, LD1RQH, and LD1RQW.
  - SVE load and broadcast quadword (scalar plus scalar): LD1ROB, LD1ROD, LD1ROH, LD1ROW, LD1RQB, LD1RQD, LD1RQH, and LD1RQW.
  - SVE load multiple structures (quadwords, scalar plus immediate): LD2Q, LD3Q, and LD4Q.
  - SVE load multiple structures (quadwords, scalar plus scalar): LD2Q, LD3Q, and LD4Q.
  - SVE load multiple structures (scalar plus immediate): LD2B, LD2D, LD2H, LD2W, LD3B, LD3D, LD3H, LD3W, LD4B, LD4D, LD4H, and LD4W.
  - SVE load multiple structures (scalar plus scalar): LD2B, LD2D, LD2H, LD2W, LD3B, LD3D, LD3H, LD3W, LD4B, LD4D, LD4H, and LD4W.
- SVE Memory - Contiguous Store and Unsized Contiguous: ST1B, ST1D, ST1H, ST1W, ST2Q, ST3Q, and ST4Q.
- SVE Memory - Contiguous Store with Immediate Offset: ST1B, ST1D, ST1H, ST1W, ST2B, ST2D, ST2H, ST2W, ST3B, ST3D, ST3H, ST3W, ST4B, ST4D, ST4H, ST4W, STNT1B, STNT1D, STNT1H, and STNT1W.
- SVE Memory - Non-temporal and Multi-register Contiguous Store: ST2B, ST2D, ST2H, ST2W, ST3B, ST3D, ST3H, ST3W, ST4B, ST4D, ST4H, ST4W, STNT1B, STNT1D, STNT1H, and STNT1W.
- SVE Memory - Non-temporal and Quadword Scatter Store: ST1Q, STNT1B, STNT1D, STNT1H, and STNT1W.
- SVE Memory - Scatter: ST1B, ST1D, ST1H, and ST1W.
- SVE Memory - Scatter with Optional Sign Extend: ST1B, ST1D, ST1H, and ST1W.
- SVE Misc: BDEP, BEXT, BGRP, EORBT, EORTB, SADDLBT, SMMLA, SSHLLB, SSHLLT, SSUBLBT, SSUBLTB, UMMLA, USHLLB, USHLLT, and USMMLA.
- SVE Multiply - Indexed:
  - SVE integer dot product (indexed): SDOT, and UDOT.
  - SVE mixed sign dot product (indexed): SUDOT, and USDOT.
  - SVE2 complex integer multiply-add (indexed): CMLA.
  - SVE2 integer multiply (indexed): MUL.
  - SVE2 integer multiply long (indexed): SMULLB, SMULLT, UMULLB, and UMULLT.
  - SVE2 integer multiply-add (indexed): MLA, and MLS.
  - SVE2 integer multiply-add long (indexed): SMLALB, SMLALT, SMLSLB, SMLSLT, UMLALB, UMLALT, UMLSLB, and UMLSLT.
  - SVE2 saturating multiply high (indexed): SQDMULH, and SQRDMULH.
  - SVE2 saturating multiply-add high (indexed): SQRDMLAH.
- SVE Permute Vector - Extract: EXT.
- SVE Permute Vector - Indexed DUP: DUP.
- SVE Permute Vector - Interleaving: TRN1, TRN2, UZP1, UZP2, ZIP1, and ZIP2.
- SVE Permute Vector - One Source Quadwords: DUPQ, and EXTQ.
- SVE Permute Vector - Predicated:
  - SVE copy SIMD&FP scalar register to vector (predicated): CPY.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SVE copy general register to vector (predicated): `CPY`.
  - SVE reverse doublewords: `REVD`.
  - SVE reverse within elements: `RBIT`, `REVB`, `REVH`, and `REVW`.
- SVE Permute Vector - Segments: `TRN1`, `TRN2`, `UZP1`, `UZP2`, `ZIP1`, and `ZIP2`.
- SVE Permute Vector - TBXQ: `TBXQ`.
- SVE Permute Vector - Three Sources TBL: `TBL`, and `TBX`.
- SVE Permute Vector - Two Sources Quadwords: `TBLQ`, `UZPQ1`, `UZPQ2`, `ZIPQ1`, and `ZIPQ2`.
- SVE Permute Vector - Two Sources TBL: `TBL`.
- SVE Permute Vector - Unpredicated:
  - SVE broadcast general register: `DUP`.
  - SVE insert SIMD&FP scalar register: `INSR`.
  - SVE insert general register: `INSR`.
  - SVE reverse vector elements: `REV`.
  - SVE unpack vector elements: `SUNPKHI`, `SUNPKLO`, `UUNPKHI`, and `UUNPKLO`.
- SVE Predicate Select: `PSEL`.
- SVE Stack Allocation: `ADDPL`, `ADDSPL`, `ADDSVL`, and `ADDVL`.
- SVE Vector Select: `SEL`.
- SVE integer clamp: `SCLAMP`, and `UCLAMP`.
- SVE two-way dot product: `SDOT`, and `UDOT`.
- SVE two-way dot product (indexed): `SDOT`, and `UDOT`.
- SVE2 Accumulate:
  - SVE2 bitwise shift and insert: `SLI`, and `SRI`.
  - SVE2 bitwise shift right and accumulate: `SRSRA`, `SSRA`, `URSRA`, and `USRA`.
  - SVE2 complex integer add: `CADD`.
  - SVE2 integer absolute difference and accumulate: `SABA`, and `UABA`.
  - SVE2 integer absolute difference and accumulate long: `SABALB`, `SABALT`, `UABALB`, and `UABALT`.
  - SVE2 integer add/subtract long with carry: `ADCLB`, `ADCLT`, `SBCLB`, and `SBCLT`.
- SVE2 Crypto Extensions: `AESD`, `AESE`, `AESIMC`, `AESMC`, `RAX1`, `SM4E`, and `SM4EKEY`.
- SVE2 Histogram Computation (Segment) and Lookup Table: `LUTI2`, and `LUTI4`.
- SVE2 Integer - Predicated:
  - SVE2 integer halving add/subtract (predicated): `SHADD`, `SHSUB`, `SHSUBR`, `UHADD`, `UHSUB` ,and `UHSUBR`.
  - SVE2 integer pairwise add and accumulate long: `SADALP`, and `UADALP`.
  - SVE2 integer pairwise arithmetic: `ADDP`, `SMAXP`, `SMINP`, `UMAXP`, and `UMINP`.
- SVE2 Integer Multiply - Unpredicated: `MUL`, `PMUL`, `SMULH`, `SQDMULH`, `SQRDMULH`, and `UMULH`.
- SVE2 Narrowing:
  - SVE2 bitwise shift right narrow: `RSHRNB`, `RSHRNT`, `SHRNB`, and `SHRNT`.
  - SVE2 integer add/subtract narrow high part: `ADDHNB`, `ADDHNT`, `RADDHNB`, `RADDHNT`, `RSUBHNB`, `RSUBHNT`, `SUBHNB`, and `SUBHNT`.
- SVE2 Widening Integer Arithmetic:
  - SVE2 integer add/subtract long: `SABDLB`, `SABDLT`, `SADDLB`, `SADDLT`, `SSUBLB`, `SSUBLT`, `UABDLB`, `UABDLT`, `UADDLB`, `UADDLT`, `USUBLB`, and `USUBLT`.
  - SVE2 integer add/subtract wide: `SADDWB`, `SADDWT`, `SSUBWB`, `SSUBWT`, `UADDWB`, `UADDWT`, `USUBWB`, and `USUBWT`.
  - SVE2 integer multiply long: `PMULLB`, `PMULLT`, `SMULLB`, `SMULLT`, `UMULLB`, and `UMULLT`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bits [23:0]

Reserved, RES0.

Accessing DIT

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DIT
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_DIT) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
```

```
MSR DIT, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_DIT) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    PSTATE.DIT = X[t, 64]<24>;
elsif PSTATE.EL == EL1 then
    PSTATE.DIT = X[t, 64]<24>;
elsif PSTATE.EL == EL2 then
    PSTATE.DIT = X[t, 64]<24>;
elsif PSTATE.EL == EL3 then
    PSTATE.DIT = X[t, 64]<24>;
```

```
MSR DIT, #<imm>
```

op0	op1	CRn	op2
0b00	0b011	0b0100	0b010

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.5 ELR_EL1, Exception Link Register (EL1)

The ELR_EL1 characteristics are:

Purpose

When taking an exception to EL1, holds the address to return to.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ELR_EL1 are UNDEFINED.

Attributes

ELR_EL1 is a 64-bit register.

Field descriptions

ADDR, bits [63:0]

Return address.

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `ELR_EL1` or `ELR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ELR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[t, 64] = NVMem[0x230];
    else
        X[t, 64] = ELR_EL1;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X[t, 64] = ELR_EL2;
    else
        X[t, 64] = ELR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL1;
```

MSR ELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪!(EffectiveHCR_EL2_NVx() IN {'x11'}) then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem[0x230] = X[t, 64];
    else
        ELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪ELIsInHost(EL2) then
        EXLOCKException();
    elsif ELIsInHost(EL2) then
        ELR_EL2 = X[t, 64];
    else
        ELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    ELR_EL1 = X[t, 64];
```

When FEAT_VHE is implemented MRS <Xt>, ELR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X[t, 64] = NVMem[0x230];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            UNDEFINED;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            X[t, 64] = ELR_EL1;
        else
            UNDEFINED;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X[t, 64] = ELR_EL1;
        else
            UNDEFINED;
```

When FEAT_VHE is implemented MSR ELR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem[0x230] = X[t, 64];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        ELR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        ELR_EL1 = X[t, 64];
    else
        UNDEFINED;
```

When FEAT_VHE is implemented MRS <Xt>, ELR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X[t, 64] = ELR_EL1;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ELR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL2;
```

When FEAT_VHE is implemented MSR ELR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& EffectiveHCR_EL2_NVx() IN {'xx1'} then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        ELR_EL1 = X[t, 64];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪then
        EXLOCKException();
    else
        ELR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    ELR_EL2 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.6 ELR_EL2, Exception Link Register (EL2)

The ELR_EL2 characteristics are:

Purpose

When taking an exception to EL2, holds the address to return to.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register ELR_EL2 bits [31:0] are architecturally mapped to AArch32 System register ELR_hyp[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ELR_EL2 are UNDEFINED.

Attributes

ELR_EL2 is a 64-bit register.

Field descriptions

ADDR, bits [63:0]

Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

When EL2 is in AArch32 Execution state and an exception is taken from EL0, EL1, or EL2 to EL3 and AArch64 execution, the upper 32-bits of ELR_EL2 are either set to 0 or hold the same value that they did before AArch32 execution. Which option is adopted is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `ELR_EL2` or `ELR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ELR_EL2
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X[t, 64] = ELR_EL1;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ELR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL2;
```

MSR ELR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪EffectiveHCR_EL2_NVx() IN {'xx1'} then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        ELR_EL1 = X[t, 64];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    else
        ELR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    ELR_EL2 = X[t, 64];
```

When FEAT_VHE is implemented MRS <Xt>, ELR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X[t, 64] = NVMem[0x230];
    else
        X[t, 64] = ELR_EL1;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X[t, 64] = ELR_EL2;
    else
        X[t, 64] = ELR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL1;
```

When FEAT_VHE is implemented MSR ELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& !(EffectiveHCR_EL2_NVx() IN {'x11'}) then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem[0x230] = X[t, 64];
    else
        ELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& ELIsInHost(EL2) then
        EXLOCKException();
    elsif ELIsInHost(EL2) then
        ELR_EL2 = X[t, 64];
    else
        ELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    ELR_EL1 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.7 ELR_EL3, Exception Link Register (EL3)

The ELR_EL3 characteristics are:

Purpose

When taking an exception to EL3, holds the address to return to.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ELR_EL3 are UNDEFINED.

Attributes

ELR_EL3 is a 64-bit register.

Field descriptions

ADDR, bits [63:0]

Return address.

An exception return from EL3 using AArch64 makes ELR_EL3 become UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ELR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL3;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ELR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    else
        ELR_EL3 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.8 FPCR, Floating-point Control Register

The FPCR characteristics are:

Purpose

Controls behavior of the floating-point instructions.

Configuration

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to nonzero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

AArch64 System register FPCR bits [26:15] are architecturally mapped to AArch32 System register FPSCR[26:15].

AArch64 System register FPCR bits [12:8] are architecturally mapped to AArch32 System register FPSCR[12:8].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to FPCR are UNDEFINED.

Attributes

FPCR is a 64-bit register.

Field descriptions

Bits [63:27]

Reserved, RES0.

AHP, bit [26]

Alternative half-precision control bit.

AHP	Meaning
0b0	IEEE half-precision format selected.
0b1	Alternative half-precision format selected.

This bit is used only for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the FEAT_FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DN, bit [25]

Default NaN use for NaN propagation.

DN	Meaning
0b0	NaN operands propagate through to the output of a floating-point operation.
0b1	Any operation involving one or more NaNs returns the Default NaN. This bit has no effect on the output of the FABS and FNEG instructions. This bit has no effect on the output of the FMAX, FMAXP, FMAXV, FMIN, FMINP, and FMINV instructions when FPCR.AH is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FZ, bit [24]

Flushing denormalized numbers to zero control bit.

FZ	Meaning
0b0	If FPCR.AH is 0, the flushing to zero of single-precision and double-precision denormalized inputs to, and outputs of, floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero. If FPCR.AH is 1, the flushing to zero of single-precision and double-precision denormalized outputs of floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero.
0b1	If FPCR.AH is 0, denormalized single-precision and double-precision inputs to, and outputs from, floating-point instructions are flushed to zero. If FPCR.AH is 1, denormalized single-precision and double-precision outputs from floating-point instructions are flushed to zero.

For more information, see ‘Flushing denormalized numbers to zero’ and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field.

RMode	Meaning
0b00	Round to Nearest (RN) mode.
0b01	Round towards Plus Infinity (RP) mode.
0b10	Round towards Minus Infinity (RM) mode.
0b11	Round towards Zero (RZ) mode.

If FPCR.AH is 1, then the following instructions use Round to Nearest mode regardless of the value of this bit:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS instructions.
- The BFCVT, BFCVTN, BFCVTN2, BFCVTNT, BFMLALB, and BFMLALT instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

This field has no function in AArch64 state, and nonzero values are ignored during execution in AArch64 state.

This field is included only for context saving and restoration of the AArch32 FPSCR.Stride field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.LEN,STRIDE as RAZ, access to this field is RAZ/WI.

FZ16, bit [19]

When FEAT_FP16 is implemented:

Flushing denormalized numbers to zero control bit on half-precision data-processing instructions.

FZ16	Meaning
`0b0`	For some instructions, this bit disables flushing to zero of inputs and outputs that are half-precision denormalized numbers.
`0b1`	Flushing denormalized numbers to zero enabled. For some instructions that do not convert a half-precision input to a higher precision output, this bit enables flushing to zero of inputs and outputs that are half-precision denormalized numbers.

For more information, see 'Flushing denormalized numbers to zero' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Len, bits [18:16]

This field has no function in AArch64 state, and nonzero values are ignored during execution in AArch64 state.

This field is included only for context saving and restoration of the AArch32 FPSCR.Len field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.LEN,STRIDE as RAZ, access to this field is RAZ/WI.

IDE, bit [15]

Input Denormal floating-point exception trap enable.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IDC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IDC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IDE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Input Denormal floating-point exceptions, access to this field is RAZ/WI.

Bit [14]

Reserved, RES0.

EBF, bit [13]

When FEAT_EBF16 is implemented:

The value of this bit controls the numeric behaviors of BFloat16 dot product calculations performed by the BFDOT, BFMMLA, BFMOPA, and BFMOPS instructions. If FEAT_SME2 is implemented, this also controls BFVDOT instruction.

When ID_AA64ISAR1_EL1.BF16 and ID_AA64ZFR0_EL1.BF16 are 0b0010, the PE supports the FPCR.EBF field. Otherwise, FPCR.EBF is RES0.

EBF	Meaning
0b0	These instructions use the standard BFloat16 behaviors: • Ignoring the FPCR.RMode control and using the rounding mode defined for BFloat16. For more information, see 'Round to Odd mode'. • Flushing denormalized inputs and outputs to zero, as if the FPCR.FZ and FPCR.FIZ controls had the value '1'. • Performing unfused multiplies and additions with intermediate rounding of all products and sums.
0b1	These instructions use the extended BFloat16 behaviors: • Supporting all four IEEE 754 rounding modes selected by the FPCR.RMode control. • Optionally, flushing denormalized inputs and outputs to zero, as governed by the FPCR.FZ and FPCR.FIZ controls. • Performing a fused two-way sum-of-products for each pair of adjacent BFloat16 elements, without intermediate rounding of the products, but rounding the single-precision sum before addition to the accumulator. • Generating the default NaN as intermediate sum-of-products when any multiplier input is a NaN, or any product is infinity × 0.0, or there are infinite products with differing signs. • Generating an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

IXE, bit [12]

Inexact floating-point exception trap enable.

IXE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IXC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IXC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IXE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Inexact floating-point exceptions, access to this field is RAZ/WI.

UFE, bit [11]

Underflow floating-point exception trap enable.

UFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.UFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs and Flush-to-zero is not enabled, the PE does not update the FPSR.UFC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.UFE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Underflow floating-point exceptions, access to this field is RAZ/WI.

OFE, bit [10]

Overflow floating-point exception trap enable.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.OFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.OFC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.OFE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Overflow floating-point exceptions, access to this field is RAZ/WI.

DZE, bit [9]

Divide by Zero floating-point exception trap enable.

DZE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.DZC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.DZC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.DZE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Divide by Zero floating-point exceptions, access to this field is RAZ/WI.

IOE, bit [8]

Invalid Operation floating-point exception trap enable.

IOE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IOC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IOC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IOE is treated as 0 for all purposes other than a direct read or write of the FPCR.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Invalid Operation floating-point exceptions, access to this field is RAZ/WI.

Bits [7:3]

Reserved, RES0.

NEP, bit [2]

When FEAT_AFP is implemented:

Controls how the output elements other than the lowest element of the vector are determined for Advanced SIMD scalar instructions.

NEP	Meaning
`0b0`	Does not affect how the output elements other than the lowest are determined for Advanced SIMD scalar instructions.

`0b1`	The output elements other than the lowest are taken from the following registers:
	• For 3-input scalar versions of the FMLA (by element) and FMLS (by element) instructions, the <Hd>, <Sd>, or <Dd> register.
	• For 3-input versions of the FMADD, FMSUB, FNMADD, and FNMSUB instructions, the <Ha>, <Sa>, or <Da> register.
	• For 2-input scalar versions of the FACGE, FACGT, FCMEQ (register), FCMGE (register), and FCMGT (register) instructions, the <Hm>, <Sm>, or <Dm> register.
	• For 2-input scalar versions of the FABD, FADD (scalar), FDIV (scalar), FMAX (scalar), FMAXNM (scalar), FMIN (scalar), FMINNM (scalar), FMUL (by element), FMUL (scalar), FMULX (by element), FMULX, FNMUL (scalar), FRECPS, FRSQRTS, and FSUB (scalar) instructions, the <Hn>, <Sn>, or <Dn> register.
	• For 1-input scalar versions of the following instructions, the <Hd>, <Sd>, or <Dd> register:
	  • The (vector) versions of the FCVTAS, FCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, and FCVTPU instructions.
	  • The (vector, fixed-point) and (vector, integer) versions of the FCVTZS, FCVTZU, SCVTF, and UCVTF instructions.
	  • The (scalar) versions of the FABS, FNEG, FRINT32X, FRINT32Z, FRINT64X, FRINT64Z, FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ, and FSQRT instructions.
	  • The (scalar, fixed-point) and (scalar, integer) versions of the SCVTF and UCVTF instructions.
	  • The BFCVT, FCVT, FCVTXN, FRECPE, FRECPX, and FRSQRTE instructions.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.NEP is treated as 0 for all purposes other than a direct read or write of the FPCR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AH, bit [1]

When FEAT_AFP is implemented:

Alternate Handling. Controls alternate handling of floating-point numbers.

The Arm architecture supports two models for handling some of the corner cases of the floating-point behaviors, such as the nature of flushing of denormalized numbers, the detection of tininess and other exceptions and a range of other behaviors. The value of the FPCR.AH bit selects between these models.

For more information on the FPCR.AH bit, see 'Flushing denormalized numbers to zero', 'Floating-point exceptions and exception traps' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FIZ, bit [0]

When FEAT_AFP is implemented:

Flush Inputs to Zero. Controls whether single-precision, double-precision and BFloat16 input operands that are denormalized numbers are flushed to zero.

FIZ	Meaning
0b0	The flushing to zero of single-precision and double-precision denormalized inputs to floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero.
0b1	Denormalized single-precision and double-precision inputs to most floating-point instructions flushed to zero.

For more information, see 'Flushing denormalized numbers to zero' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing FPCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FPCR
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif ELIsInHost(EL0) && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPCR;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPCR;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPCR;
```

```
MSR FPCR, <Xt>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif ELIsInHost(EL0) && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t, 64];
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.9 FPMR, Floating-point Mode Register

The FPMR characteristics are:

Purpose

Controls behaviors of the FP8 instructions.

Configuration

A direct or indirect read of this register occurs in program order relative to a direct write of this register without explicit synchronization.

On entry to or exit from Streaming SVE mode, FPMR is set to 0.

This register is present only when FEAT_FPMR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to FPMR are UNDEFINED.

Attributes

FPMR is a 64-bit register.

Field descriptions

Bits [63:38]

Reserved, RES0.

LSCALE2, bits [37:32]

Downscaling value for instructions that convert the second FP8 input data stream to other floating-point formats.

This value is an unsigned integer that is subtracted from the result exponent.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSCALE, bits [31:24]

Scaling value for instructions that convert other floating-point formats to an FP8 format.

This value is a signed integer that is added to the operand exponent.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [23]

Reserved, RES0.

LSCALE, bits [22:16]

Downscaling value.

This value is an unsigned integer that is subtracted from:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The product or the sum-of-products exponent, for multiplication instructions with FP8 operands.
- The result exponent, for instructions that convert the first FP8 input data stream to other floating-point formats.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OSC, bit [15]

Overflow saturation for FP8 convert instructions. Specifies the result when a floating-point Overflow exception is detected.

OSC	Meaning
`0b0`	Infinity or NaN is generated.
`0b1`	Maximum normal number is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OSM, bit [14]

Overflow saturation for FP8 multiplication instructions. Specifies the result when a floating-point Overflow exception is detected.

OSM	Meaning
`0b0`	Infinity is generated.
`0b1`	Maximum normal number is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [13:9]

Reserved, RES0.

F8D, bits [8:6]

Destination result format for instructions that convert other floating-point values to an FP8 format.

F8D	Meaning
`0b000`	OFP8 E5M2 format.
`0b001`	OFP8 E4M3 format.

All other values are reserved.

Reserved values identify an unsupported format and behave as described in Reserved values in System and memory-mapped registers and translation table entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additionally, FP8 instructions are permitted to set an FP8 result with an unsupported format to `0xFF` and signal an Invalid Operation floating-point exception.

It is software's responsibility to check that a format value is supported in ID_AA64FPFR0_EL1[7:0], before writing it to this field.

For more information about the FP8 formats, see the *OCP 8-bit Floating Point Specification (OFP8)*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F8S2, bits [5:3]

Second FP8 input data stream format for multiplication instructions with FP8 operands, and the corresponding instructions that convert an FP8 format to other floating-point formats.

F8S2	Meaning
`0b000`	OFP8 E5M2 format.
`0b001`	OFP8 E4M3 format.

All other values are reserved.

Reserved values identify an unsupported format and behave as described in Reserved values in System and memory-mapped registers and translation table entries.

Additionally FP8 instructions are permitted to treat FP8 input values with an unsupported format as a signaling NaN.

It is software's responsibility to check that a format value is supported in ID_AA64FPFR0_EL1[7:0], before writing it to this field.

For more information about the FP8 formats, see the *OCP 8-bit Floating Point Specification (OFP8)*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F8S1, bits [2:0]

First FP8 input data stream format for multiplication instructions with FP8 operands, and the corresponding instructions that convert an FP8 format to other floating-point formats.

F8S1	Meaning
`0b000`	OFP8 E5M2 format.
`0b001`	OFP8 E4M3 format.

All other values are reserved.

Reserved values identify an unsupported format and behave as described in Reserved values in System and memory-mapped registers and translation table entries.

Additionally FP8 instructions are permitted to treat FP8 input values with an unsupported format as a signaling NaN.

It is software's responsibility to check that a format value is supported in ID_AA64FPFR0_EL1[7:0], before writing it to this field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about the FP8 formats, see the *OCP 8-bit Floating Point Specification (OFP8)*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FPMR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FPMR
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_FPMR) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3.EnFPM == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !ELIsInHost(EL0) && SCTLR_EL1.EnFPM == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.EnFPM == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && ((HaveEL(EL3) && SCR_EL3.HXEn == '0') || HCRX_EL2.EnFPM ==
     ↪'0') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnFPM == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif ELIsInHost(EL0) && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPMR;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3.EnFPM == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL0) && ((HaveEL(EL3) && SCR_EL3.HXEn == '0') || HCRX_EL2.EnFPM ==
        ↪'0') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnFPM == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPMR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3.EnFPM == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.EnFPM == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPMR;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPMR;
```

```
MSR FPMR, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_FPMR) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3.EnFPM == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif !ELIsInHost(EL0) && SCTLR_EL1.EnFPM == '0' then
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL1, 0x18);
        elsif ELIsInHost(EL0) && SCTLR_EL2.EnFPM == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && !ELIsInHost(EL0) && ((HaveEL(EL3) && SCR_EL3.HXEn == '0') || HCRX_EL2.EnFPM ==
             ↪'0') then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3.EnFPM == '0' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        elsif !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                AArch64.SystemAccessTrap(EL2, 0x00);
            else
                AArch64.SystemAccessTrap(EL1, 0x07);
        elsif ELIsInHost(EL0) && CPTR_EL2.FPEN != '11' then
            AArch64.SystemAccessTrap(EL2, 0x07);
        elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
            AArch64.SystemAccessTrap(EL2, 0x07);
        elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
            AArch64.SystemAccessTrap(EL2, 0x07);
        elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x07);
        else
            FPMR = X[t, 64];
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3.EnFPM == '0' then
            UNDEFINED;
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
            UNDEFINED;
        elsif EL2Enabled() && !ELIsInHost(EL0) && ((HaveEL(EL3) && SCR_EL3.HXEn == '0') || HCRX_EL2.EnFPM ==
             ↪'0') then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3.EnFPM == '0' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        elsif CPACR_EL1.FPEN IN {'x0'} then
            AArch64.SystemAccessTrap(EL1, 0x07);
        elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
            AArch64.SystemAccessTrap(EL2, 0x07);
        elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
            AArch64.SystemAccessTrap(EL2, 0x07);
        elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x07);
        else
            FPMR = X[t, 64];
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3.EnFPM == '0' then
            UNDEFINED;
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && SCR_EL3.EnFPM == '0' then
            if EL3SDDUndef() then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPMR = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPMR = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.10 FPSR, Floating-point Status Register

The FPSR characteristics are:

Purpose

Provides floating-point system status information.

Configuration

On entry to or exit from Streaming SVE mode, FPSR.{IOC, DZC, OFC, UFC, IXC, IDC, QC} are set to 1 and the remaining bits are set to 0.

AArch64 System register FPSR bits [31:27] are architecturally mapped to AArch32 System register FPSCR[31:27].

AArch64 System register FPSR bit [7] is architecturally mapped to AArch32 System register FPSCR[7].

AArch64 System register FPSR bits [4:0] are architecturally mapped to AArch32 System register FPSCR[4:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to FPSR are UNDEFINED.

Attributes

FPSR is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

N, bit [31]

When FEAT_AA32 is implemented and FEAT_FP is implemented:

Negative condition flag for AArch32 floating-point comparison operations.

Note

AArch64 floating-point comparisons set the PSTATE.N flag instead.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Z, bit [30]

When FEAT_AA32 is implemented and FEAT_FP is implemented:

Zero condition flag for AArch32 floating-point comparison operations.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

AArch64 floating-point comparisons set the PSTATE.Z flag instead.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

C, bit [29]

When FEAT_AA32 is implemented and FEAT_FP is implemented:

Carry condition flag for AArch32 floating-point comparison operations.

Note

AArch64 floating-point comparisons set the PSTATE.C flag instead.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

V, bit [28]

When FEAT_AA32 is implemented and FEAT_FP is implemented:

Overflow condition flag for AArch32 floating-point comparison operations.

Note

AArch64 floating-point comparisons set the PSTATE.V flag instead.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [26:8]

Reserved, RES0.

IDC, bit [7]

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IDE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IDE is 0.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXC, bit [4]

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IXE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IXE is 0.

The criteria for the Inexact floating-point exception to occur are affected by whether denormalized numbers are flushed to zero and by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFC, bit [3]

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.UFE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.UFE is 0 or if flushing denormalized numbers to zero is enabled.

The criteria for the Underflow floating-point exception to occur are affected by whether denormalized numbers are flushed to zero and by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFC, bit [2]

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.OFE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.OFE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZC, bit [1]

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.DZE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.DZE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOC, bit [0]

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IOE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IOE is 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The criteria for the Invalid Operation floating-point exception to occur are affected by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FPSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FPSR
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif ELIsInHost(EL0) && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPSR;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPSR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPSR;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPSR;
```

```
MSR FPSR, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif ELIsInHost(EL0) && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPSR = X[t, 64];
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPSR = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.TFP == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        UNDEFINED;
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPSR = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPSR = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.11 NZCV, Condition Flags

The NZCV characteristics are:

Purpose

Allows access to the condition flags.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to NZCV are UNDEFINED.

Attributes

NZCV is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative condition flag. Set to 1 if the result of the last flag-setting instruction was negative.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Bits [27:0]

Reserved, RES0.

Accessing NZCV

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, NZCV
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    X[t, 64] = Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
```

```
MSR NZCV, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL1 then
    PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL2 then
    PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL3 then
    PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.12 PAN, Privileged Access Never

The PAN characteristics are:

Purpose

Allows access to the Privileged Access Never bit.

Configuration

This register is present only when FEAT_PAN is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PAN are UNDEFINED.

Attributes

PAN is a 64-bit register.

Field descriptions

Bits [63:23]

Reserved, RES0.

PAN, bit [22]

Privileged Access Never.

PAN	Meaning
0b0	Privileged reads and write are not disabled by this mechanism.
0b1	Disables privileged read and write accesses to addresses accessible at EL0 for an enabled stage 1 translation regime that defines the EL0 permissions.

The value of this bit is usually preserved on taking an exception, except in the following situations:

- When the target of the exception is EL1, and the value of the SCTLR_EL1.SPAN bit is 0, this bit is set to 1.
- When the target of the exception is EL2, the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and the value of the SCTLR_EL2.SPAN bit is 0, this bit is set to 1.

Bits [21:0]

Reserved, RES0.

Accessing PAN

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, PAN
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_PAN) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(41):PSTATE.PAN:Zeros(22);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(41):PSTATE.PAN:Zeros(22);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(41):PSTATE.PAN:Zeros(22);
```

```
MSR PAN, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_PAN) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.PAN = X[t, 64]<22>;
elsif PSTATE.EL == EL2 then
    PSTATE.PAN = X[t, 64]<22>;
elsif PSTATE.EL == EL3 then
    PSTATE.PAN = X[t, 64]<22>;
```

```
MSR PAN, #<imm>
```

op0	op1	CRn	op2
0b00	0b000	0b0100	0b100

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.13 PM, Profiling Exception Mask

The PM characteristics are:

Purpose

Allows access to the Profiling exception Mask bit.

Configuration

This register is present only when FEAT_EBEP is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PM are UNDEFINED.

Attributes

PM is a 64-bit register.

Field descriptions

Bits [63:33]

Reserved, RES0.

PM, bit [32]

PMU Exception Mask.

PM	Meaning
0b0	Does not cause the Profiling exception to be masked.
0b1	Causes the Profiling exception to be masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

Accessing PM

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PM
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_EBEP) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(31):PSTATE.PM:Zeros(32);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(31):PSTATE.PM:Zeros(32);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(31):PSTATE.PM:Zeros(32);
```

```
MSR PM, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_EBEP) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.PM = X[t, 64]<32>;
elsif PSTATE.EL == EL2 then
    PSTATE.PM = X[t, 64]<32>;
elsif PSTATE.EL == EL3 then
    PSTATE.PM = X[t, 64]<32>;
```

```
MSR PM, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b001	0b0100	0b001x	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.14 SP_EL0, Stack Pointer (EL0)

The SP_EL0 characteristics are:

Purpose

Holds the stack pointer associated with EL0. At higher Exception levels, this is used as the current stack pointer when the value of SPSel.SP is 0.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SP_EL0 are UNDEFINED.

Attributes

SP_EL0 is a 64-bit register.

Field descriptions

StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL0

When the value of PSTATE.SP is 0, this register is accessible at all Exception levels as the current stack pointer.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SP_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        X[t, 64] = SP_EL0;
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        UNDEFINED;
    else
        X[t, 64] = SP_EL0;
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        X[t, 64] = SP_EL0;
```

```
MSR SP_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.15 SP_EL1, Stack Pointer (EL1)

The SP_EL1 characteristics are:

Purpose

Holds the stack pointer associated with EL1. The value of SPSel.SP determines the current stack pointer.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SP_EL1 are UNDEFINED.

Attributes

SP_EL1 is a 64-bit register.

Field descriptions

StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL1

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL1 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SP_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[t, 64] = NVMem[0x240];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SP_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SP_EL1;
```

```
MSR SP_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem[0x240] = X[t, 64];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SP_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    SP_EL1 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.16 SP_EL2, Stack Pointer (EL2)

The SP_EL2 characteristics are:

Purpose

Holds the stack pointer associated with EL2. The value of SPSel.SP determines the current stack pointer.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SP_EL2 are UNDEFINED.

Attributes

SP_EL2 is a 64-bit register.

Field descriptions

StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL2

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL2 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SP_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SP_EL2;
```

```
MSR SP_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SP_EL2 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.17 SP_EL3, Stack Pointer (EL3)

The SP_EL3 characteristics are:

Purpose

Holds the stack pointer associated with EL3. The value of SPSel.SP determines the current stack pointer.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SP_EL3 are UNDEFINED.

Attributes

SP_EL3 is a 64-bit register.

Field descriptions

StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL3

This register is not accessible using MRS and MSR instructions.

When the value of SPSel.SP is 1, this register is accessible at EL3 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.18 SPSel, Stack Pointer Select

The SPSel characteristics are:

Purpose

Allows the Stack Pointer to be selected between SP_EL0 and SP_ELx.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSel are UNDEFINED.

Attributes

SPSel is a 64-bit register.

Field descriptions

Bits [63:1]

Reserved, RES0.

SP, bit [0]

Stack pointer to use. Possible values of this bit are:

SP	Meaning
0b0	Use SP_EL0 at all Exception levels.
0b1	Use SP_ELx for Exception level ELx. When FEAT_NMI is implemented and SCTLR_ELx.SPINTMASK is 1, if execution is at ELx, an IRQ or FIQ interrupt that is targeted to ELx is masked regardless of any indication of Superpriority.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

Accessing SPSel

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPSel
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(63):PSTATE.SP;
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(63):PSTATE.SP;
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(63):PSTATE.SP;
```

```
MSR SPSel, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.SP = X[t, 64]<0>;
elsif PSTATE.EL == EL2 then
    PSTATE.SP = X[t, 64]<0>;
elsif PSTATE.EL == EL3 then
    PSTATE.SP = X[t, 64]<0>;
```

```
MSR SPSel, #<imm>
```

op0	op1	CRn	op2
0b00	0b000	0b0100	0b101

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.19 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

Purpose

Holds the saved process state when an exception is taken to Abort mode.

Configuration

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

AArch64 System register SPSR_abt bits [31:0] are architecturally mapped to AArch32 System register SPSR_abt[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_abt are UNDEFINED.

Attributes

SPSR_abt is a 64-bit register.

Field descriptions

When FEAT_AA32EL1 is not implemented:

Bits [63:0]

Reserved, RES0.

Otherwise:

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Abort mode, and copied to PSTATE.N on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Abort mode, and copied to PSTATE.Z on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Abort mode, and copied to PSTATE.C on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Abort mode, and copied to PSTATE.V on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Abort mode, and copied to PSTATE.Q on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Abort mode, and copied to PSTATE.IT on executing an exception return operation in Abort mode.

On executing an exception return operation in Abort mode, SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Abort mode, and copied to PSTATE.SSBS on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Abort mode, and copied to PSTATE.PAN on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Abort mode, and copied to PSTATE.DIT on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Abort mode, and copied to PSTATE.IL on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Abort mode, and copied to PSTATE.GE on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Abort mode, and copied to PSTATE.E on executing an exception return operation in Abort mode.

If the implementation does not support big-endian operation, SPSR_abt.E is RES0. If the implementation does not support little-endian operation, SPSR_abt.E is RES1. On executing an exception return operation in Abort mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_abt.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_abt.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Abort mode, and copied to PSTATE.A on executing an exception return operation in Abort mode.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Abort mode, and copied to PSTATE.I on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Abort mode, and copied to PSTATE.F on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Abort mode, and copied to PSTATE.T on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Abort mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Abort mode.

M[4:0]	Meaning
`0b10000`	User.
`0b10001`	FIQ.
`0b10010`	IRQ.
`0b10011`	Supervisor.
`0b10111`	Abort.
`0b11011`	Undefined.
`0b11111`	System.

Other values are reserved. If SPSR_abt.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Abort mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_abt

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_abt
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_abt;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_abt;
```

```
MSR SPSR_abt, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_abt = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_abt = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.20 SPSR_EL1, Saved Program Status Register (EL1)

The SPSR_EL1 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL1.

Configuration

AArch64 System register SPSR_EL1 bits [31:0] are architecturally mapped to AArch32 System register SPSR_svc[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_EL1 are UNDEFINED.

Attributes

SPSR_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented and exception taken from AArch32 state:

An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL1, and copied to PSTATE.UINJ on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:34]

Reserved, RES0.

PPEND, bit [33]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SEBEP is implemented:

PMU Profiling exception pending bit. Set to the value of PSTATE.PPEND on taking an exception to EL1, and conditionally copied to PSTATE.PPEND on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL1, and copied to PSTATE.Q on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to EL1, and copied to PSTATE.IT on executing an exception return operation in EL1.

On executing an exception return operation in EL1, SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL1, and copied to PSTATE.GE on executing an exception return operation in EL1.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL1, and copied to PSTATE.E on executing an exception return operation in EL1.

If the implementation does not support big-endian operation, SPSR_EL1.E is RES0. If the implementation does not support little-endian operation, SPSR_EL1.E is RES1. On executing an exception return operation in EL1, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL1.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL1.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL1, and copied to PSTATE.T on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to `0b1`, the value of PSTATE.nRW, on taking an exception to EL1 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

M[4]	Meaning
`0b1`	AArch32 execution state.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL1, and copied to PSTATE.M[3:0] on executing an exception return operation in EL1.

M[3:0]	Meaning
0b0000	User.
0b0001	FIQ.
0b0010	IRQ.
0b0011	Supervisor.
0b0111	Abort.
0b1011	Undefined.
0b1111	System.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:

An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL1, and copied to PSTATE.UINJ on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

PACM, bit [35]

When FEAT_PAuth_LR is implemented:

PACM. Set to the value of PSTATE.PACM on taking an exception to EL1, and copied to PSTATE.PACM on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLOCK, bit [34]

When FEAT_GCS is implemented:

Exception return state lock. Set to the value of PSTATE.EXLOCK on taking an exception to EL1, and copied to PSTATE.EXLOCK on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PPEND, bit [33]

When FEAT_SEBEP is implemented:

PMU Profiling exception pending bit. Set to the value of PSTATE.PPEND on taking an exception to EL1, and conditionally copied to PSTATE.PPEND on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PM, bit [32]

When FEAT_EBEP is implemented:

Profiling exception mask bit. Set to the value of PSTATE.PM on taking an exception to EL1, and copied to PSTATE.PM on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When FEAT_MTE is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL1, and copied to PSTATE.TCO on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL1, and copied to PSTATE.UAO on executing an exception return operation in EL1.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

ALLINT, bit [13]

When FEAT_NMI is implemented:

All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on taking an exception to EL1, and copied to PSTATE.ALLINT on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When FEAT_BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL1, and copied to PSTATE.BTYPE on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL1, and copied to PSTATE.D on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on taking an exception to EL1 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4]	Meaning
0b0	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning
0b0000	EL0.
0b0100	EL1 with SP_EL0 (EL1t).
0b0101	EL1 with SP_EL1 (EL1h).
0b1000	EL1 with SP_EL0 (EL1t) and either HCR_EL2.{NV, NV1} is {1,0} or HCR_EL2.{NV, NV2} is {1,1}.
0b1001	EL1 with SP_EL1 (EL1h) and either HCR_EL2.{NV, NV1} is {1,0} or HCR_EL2.{NV, NV2} is {1,1}.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2]: On an exception to EL1:
  - If the exception is taken from EL0:
    - M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1.
  - If the exception is taken from EL1:
    - If the Effective value of HCR_EL2.{NV, NV1} is not {1,0} and the Effective value of HCR_EL2.{NV, NV2} is not {1,1}, then M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1.
    - If the Effective value of HCR_EL2.{NV, NV1} is {1,0} or if the Effective value of HCR_EL2.{NV, NV2} is {1,1}, then M[3:2] is set to 0b10.
  - M[3:2] is copied to PSTATE.EL on executing a legal exception return operation in EL1.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL1 and copied to PSTATE.SP on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name SPSR_EL1 or SPSR_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X[t, 64] = NVMem[0x160];
    else
        X[t, 64] = SPSR_EL1;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X[t, 64] = SPSR_EL2;
    else
        X[t, 64] = SPSR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_EL1;
```

```
MSR SPSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪!(EffectiveHCR_EL2_NVx() IN {'x11'}) then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem[0x160] = X[t, 64];
    else
        SPSR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪ELIsInHost(EL2) then
        EXLOCKException();
    elsif ELIsInHost(EL2) then
        SPSR_EL2 = X[t, 64];
    else
        SPSR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_EL1 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_VHE is implemented MRS <Xt>, SPSR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X[t, 64] = NVMem[0x160];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X[t, 64] = SPSR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X[t, 64] = SPSR_EL1;
    else
        UNDEFINED;
```

When FEAT_VHE is implemented MSR SPSR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem[0x160] = X[t, 64];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        SPSR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        SPSR_EL1 = X[t, 64];
    else
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_VHE is implemented MRS <Xt>, SPSR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X[t, 64] = SPSR_EL1;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_EL2;
```

When FEAT_VHE is implemented MSR SPSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& EffectiveHCR_EL2_NVx() IN {'xx1'} then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        SPSR_EL1 = X[t, 64];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪then
        EXLOCKException();
    else
        SPSR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_EL2 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.21 SPSR_EL2, Saved Program Status Register (EL2)

The SPSR_EL2 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL2.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register SPSR_EL2 bits [31:0] are architecturally mapped to AArch32 System register SPSR_hyp[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_EL2 are UNDEFINED.

Attributes

SPSR_EL2 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented and exception taken from AArch32 state:

An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL2, and copied to PSTATE.UINJ on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:34]

Reserved, RES0.

PPEND, bit [33]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SEBEP is implemented:

PMU Profiling exception pending bit. Set to the value of PSTATE.PPEND on taking an exception to EL2, and conditionally copied to PSTATE.PPEND on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL2, and copied to PSTATE.Q on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to EL2, and copied to PSTATE.IT on executing an exception return operation in EL2.

On executing an exception return operation in EL2, SPSR_EL2.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL2, and copied to PSTATE.GE on executing an exception return operation in EL2.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL2, and copied to PSTATE.E on executing an exception return operation in EL2.

If the implementation does not support big-endian operation, SPSR_EL2.E is RES0. If the implementation does not support little-endian operation, SPSR_EL2.E is RES1. On executing an exception return operation in EL2, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL2.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL2.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL2, and copied to PSTATE.T on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to `0b1`, the value of PSTATE.nRW, on taking an exception to EL2 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

M[4]	Meaning
`0b1`	AArch32 execution state.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL2, and copied to PSTATE.M[3:0] on executing an exception return operation in EL2.

M[3:0]	Meaning
0b0000	User.
0b0001	FIQ.
0b0010	IRQ.
0b0011	Supervisor.
0b0111	Abort.
0b1010	Hyp.
0b1011	Undefined.
0b1111	System.

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:

An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL2, and copied to PSTATE.UINJ on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

PACM, bit [35]

When FEAT_PAuth_LR is implemented:

PACM. Set to the value of PSTATE.PACM on taking an exception to EL2, and copied to PSTATE.PACM on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLOCK, bit [34]

When FEAT_GCS is implemented:

Exception return state lock. Set to the value of PSTATE.EXLOCK on taking an exception to EL2, and copied to PSTATE.EXLOCK on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PPEND, bit [33]

When FEAT_SEBEP is implemented:

PMU Profiling exception pending bit. Set to the value of PSTATE.PPEND on taking an exception to EL2, and conditionally copied to PSTATE.PPEND on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PM, bit [32]

When FEAT_EBEP is implemented:

Profiling exception mask bit. Set to the value of PSTATE.PM on taking an exception to EL2, and copied to PSTATE.PM on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When FEAT_MTE is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL2, and copied to PSTATE.TCO on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL2, and copied to PSTATE.UAO on executing an exception return operation in EL2.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

ALLINT, bit [13]

When FEAT_NMI is implemented:

All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on taking an exception to EL2, and copied to PSTATE.ALLINT on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When FEAT_BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL2, and copied to PSTATE.BTYPE on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL2, and copied to PSTATE.D on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on taking an exception to EL2 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4]	Meaning
0b0	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning
0b0000	EL0.
0b0100	EL1 with SP_EL0 (EL1t).
0b0101	EL1 with SP_EL1 (EL1h).
0b1000	EL2 with SP_EL0 (EL2t).
0b1001	EL2 with SP_EL2 (EL2h).

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL2 and copied to PSTATE.EL on executing an exception return operation in EL2.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL2 and copied to PSTATE.SP on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `SPSR_EL2` or `SPSR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPSR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X[t, 64] = SPSR_EL1;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_EL2;
```

MSR SPSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪EffectiveHCR_EL2_NVx() IN {'xx1'} then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        SPSR_EL1 = X[t, 64];
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    else
        SPSR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_EL2 = X[t, 64];
```

When FEAT_VHE is implemented MRS <Xt>, SPSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X[t, 64] = NVMem[0x160];
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[t, 64] = SPSR_EL1;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X[t, 64] = SPSR_EL2;
    else
        X[t, 64] = SPSR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_EL1;
```

When FEAT_VHE is implemented MSR SPSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& !(EffectiveHCR_EL2_NVx() IN {'x11'}) then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem[0x160] = X[t, 64];
    else
        SPSR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& ELIsInHost(EL2) then
        EXLOCKException();
    elsif ELIsInHost(EL2) then
        SPSR_EL2 = X[t, 64];
    else
        SPSR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_EL1 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.22 SPSR_EL3, Saved Program Status Register (EL3)

The SPSR_EL3 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_EL3 are UNDEFINED.

Attributes

SPSR_EL3 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented and exception taken from AArch32 state:

An exception return from EL3 using AArch64 makes SPSR_EL3 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL3, and copied to PSTATE.UINJ on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:34]

Reserved, RES0.

PPEND, bit [33]

When FEAT_SEBEP is implemented:

PMU Profiling exception pending bit. Set to the value of PSTATE.PPEND on taking an exception to EL3, and conditionally copied to PSTATE.PPEND on executing an exception return operation in EL3.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL3, and copied to PSTATE.Q on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to EL3, and copied to PSTATE.IT on executing an exception return operation in EL3.

On executing an exception return operation in EL3, SPSR_EL3.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL3, and copied to PSTATE.DIT on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL3, and copied to PSTATE.GE on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL3, and copied to PSTATE.E on executing an exception return operation in EL3.

If the implementation does not support big-endian operation, SPSR_EL3.E is RES0. If the implementation does not support little-endian operation, SPSR_EL3.E is RES1. On executing an exception return operation in EL3, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL3.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL3.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL3, and copied to PSTATE.T on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to `0b1`, the value of PSTATE.nRW, on taking an exception to EL3 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

M[4]	Meaning
`0b1`	AArch32 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL3, and copied to PSTATE.M[3:0] on executing an exception return operation in EL3.

M[3:0]	Meaning	Applies when
0b0000	User.	
0b0001	FIQ.	
0b0010	IRQ.	
0b0011	Supervisor.	
0b0110	Monitor.	FEAT_EL3 is implemented
0b0111	Abort.	
0b1010	Hyp.	
0b1011	Undefined.	
0b1111	System.	

Other values are reserved. If SPSR_EL3.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:

An exception return from EL3 using AArch64 makes SPSR_EL3 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL3, and copied to PSTATE.UINJ on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

PACM, bit [35]

When FEAT_PAuth_LR is implemented:

PACM. Set to the value of PSTATE.PACM on taking an exception to EL3, and copied to PSTATE.PACM on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLOCK, bit [34]

When FEAT_GCS is implemented:

Exception return state lock. Set to the value of PSTATE.EXLOCK on taking an exception to EL3, and copied to PSTATE.EXLOCK on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PPEND, bit [33]

When FEAT_SEBEP is implemented:

PMU Profiling exception pending bit. Set to the value of PSTATE.PPEND on taking an exception to EL3, and conditionally copied to PSTATE.PPEND on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PM, bit [32]

When FEAT_EBEP is implemented:

Profiling exception mask bit. Set to the value of PSTATE.PM on taking an exception to EL3, and copied to PSTATE.PM on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When FEAT_MTE is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL3, and copied to PSTATE.TCO on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL3, and copied to PSTATE.DIT on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL3, and copied to PSTATE.UAO on executing an exception return operation in EL3.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

ALLINT, bit [13]

When FEAT_NMI is implemented:

All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on taking an exception to EL3, and copied to PSTATE.ALLINT on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When FEAT_BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL3, and copied to PSTATE.BTYPE on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL3, and copied to PSTATE.D on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on taking an exception to EL3 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4]	Meaning
0b0	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning	Applies when
0b0000	EL0.	
0b0100	EL1 with SP_EL0 (EL1t).	
0b0101	EL1 with SP_EL1 (EL1h).	
0b1000	EL2 with SP_EL0 (EL2t).	
0b1001	EL2 with SP_EL2 (EL2h).	
0b1100	EL3 with SP_EL0 (EL3t).	FEAT_EL3 is implemented
0b1101	EL3 with SP_EL3 (EL3h).	FEAT_EL3 is implemented

Other values are reserved. If SPSR_EL3.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL3 and copied to PSTATE.EL on executing an exception return operation in EL3.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL3 and copied to PSTATE.SP on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPSR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_EL3;
```

```
MSR SPSR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    else
        SPSR_EL3 = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.23 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configuration

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

AArch64 System register SPSR_fiq bits [31:0] are architecturally mapped to AArch32 System register SPSR_fiq[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_fiq are UNDEFINED.

Attributes

SPSR_fiq is a 64-bit register.

Field descriptions

When FEAT_AA32EL1 is not implemented:

Bits [63:0]

Reserved, RES0.

Otherwise:

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to FIQ mode, and copied to PSTATE.N on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to FIQ mode, and copied to PSTATE.Z on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to FIQ mode, and copied to PSTATE.C on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to FIQ mode, and copied to PSTATE.V on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to FIQ mode, and copied to PSTATE.Q on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to FIQ mode, and copied to PSTATE.IT on executing an exception return operation in FIQ mode.

On executing an exception return operation in FIQ mode, SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to FIQ mode, and copied to PSTATE.SSBS on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to FIQ mode, and copied to PSTATE.PAN on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to FIQ mode, and copied to PSTATE.DIT on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to FIQ mode, and copied to PSTATE.IL on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to FIQ mode, and copied to PSTATE.GE on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to FIQ mode, and copied to PSTATE.E on executing an exception return operation in FIQ mode.

If the implementation does not support big-endian operation, SPSR_fiq.E is RES0. If the implementation does not support little-endian operation, SPSR_fiq.E is RES1. On executing an exception return operation in FIQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_fiq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_fiq.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to FIQ mode, and copied to PSTATE.A on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to FIQ mode, and copied to PSTATE.I on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to FIQ mode, and copied to PSTATE.F on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to FIQ mode, and copied to PSTATE.T on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to FIQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in FIQ mode.

M[4:0]	Meaning
`0b10000`	User.
`0b10001`	FIQ.
`0b10010`	IRQ.
`0b10011`	Supervisor.
`0b10111`	Abort.
`0b11011`	Undefined.
`0b11111`	System.

Other values are reserved. If SPSR_fiq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in FIQ mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_fiq

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_fiq
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_fiq;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_fiq;
```

```
MSR SPSR_fiq, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_fiq = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_fiq = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.24 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configuration

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

AArch64 System register SPSR_irq bits [31:0] are architecturally mapped to AArch32 System register SPSR_irq[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_irq are UNDEFINED.

Attributes

SPSR_irq is a 64-bit register.

Field descriptions

When FEAT_AA32EL1 is not implemented:

Bits [63:0]

Reserved, RES0.

Otherwise:

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to IRQ mode, and copied to PSTATE.N on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to IRQ mode, and copied to PSTATE.Z on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to IRQ mode, and copied to PSTATE.C on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to IRQ mode, and copied to PSTATE.V on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to IRQ mode, and copied to PSTATE.Q on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to IRQ mode, and copied to PSTATE.IT on executing an exception return operation in IRQ mode.

On executing an exception return operation in IRQ mode, SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to IRQ mode, and copied to PSTATE.SSBS on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to IRQ mode, and copied to PSTATE.PAN on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to IRQ mode, and copied to PSTATE.DIT on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to IRQ mode, and copied to PSTATE.IL on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to IRQ mode, and copied to PSTATE.GE on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to IRQ mode, and copied to PSTATE.E on executing an exception return operation in IRQ mode.

If the implementation does not support big-endian operation, SPSR_irq.E is RES0. If the implementation does not support little-endian operation, SPSR_irq.E is RES1. On executing an exception return operation in IRQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_irq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_irq.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to IRQ mode, and copied to PSTATE.A on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to IRQ mode, and copied to PSTATE.I on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to IRQ mode, and copied to PSTATE.F on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to IRQ mode, and copied to PSTATE.T on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to IRQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in IRQ mode.

M[4:0]	Meaning
`0b10000`	User.
`0b10001`	FIQ.
`0b10010`	IRQ.
`0b10011`	Supervisor.
`0b10111`	Abort.
`0b11011`	Undefined.
`0b11111`	System.

Other values are reserved. If SPSR_irq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in IRQ mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_irq

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_irq
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_irq;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_irq;
```

```
MSR SPSR_irq, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_irq = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_irq = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.25 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

Purpose

Holds the saved process state when an exception is taken to Undefined mode.

Configuration

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

AArch64 System register SPSR_und bits [31:0] are architecturally mapped to AArch32 System register SPSR_und[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_und are UNDEFINED.

Attributes

SPSR_und is a 64-bit register.

Field descriptions

When FEAT_AA32EL1 is not implemented:

Bits [63:0]

Reserved, RES0.

Otherwise:

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Undefined mode, and copied to PSTATE.N on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Undefined mode, and copied to PSTATE.Z on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Undefined mode, and copied to PSTATE.C on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Undefined mode, and copied to PSTATE.V on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Undefined mode, and copied to PSTATE.Q on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Undefined mode, and copied to PSTATE.IT on executing an exception return operation in Undefined mode.

On executing an exception return operation in Undefined mode, SPSR_und.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Undefined mode, and copied to PSTATE.SSBS on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Undefined mode, and copied to PSTATE.PAN on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Undefined mode, and copied to PSTATE.DIT on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Undefined mode, and copied to PSTATE.IL on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Undefined mode, and copied to PSTATE.GE on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Undefined mode, and copied to PSTATE.E on executing an exception return operation in Undefined mode.

If the implementation does not support big-endian operation, SPSR_und.E is RES0. If the implementation does not support little-endian operation, SPSR_und.E is RES1. On executing an exception return operation in Undefined mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_und.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_und.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Undefined mode, and copied to PSTATE.A on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Undefined mode, and copied to PSTATE.I on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Undefined mode, and copied to PSTATE.F on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Undefined mode, and copied to PSTATE.T on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Undefined mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Undefined mode.

M[4:0]	Meaning
`0b10000`	User.
`0b10001`	FIQ.
`0b10010`	IRQ.
`0b10011`	Supervisor.
`0b10111`	Abort.
`0b11011`	Undefined.
`0b11111`	System.

Other values are reserved. If SPSR_und.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Undefined mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_und

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_und
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_und;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_und;
```

```
MSR SPSR_und, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_und = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_und = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.26 SSBS, Speculative Store Bypass Safe

The SSBS characteristics are:

Purpose

Allows access to the Speculative Store Bypass Safe bit.

Configuration

This register is present only when FEAT_SSBS2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SSBS are UNDEFINED.

Attributes

SSBS is a 64-bit register.

Field descriptions

Bits [63:13]

Reserved, RES0.

SSBS, bit [12]

Speculative Store Bypass Safe.

Prohibits speculative loads or stores which might practically allow a cache timing side channel.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

SSBS	Meaning
0b0	Hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.
0b1	When the value of PSTATE.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

The value of this bit is set to the value in the SCTLR_ELx.DSSBS field on taking an exception to ELx.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [11:0]

Reserved, RES0.

Accessing SSBS

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SSBS
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b110

```
if !(IsFeatureImplemented(FEAT_SSBS2) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
```

```
MSR SSBS, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b110

```
if !(IsFeatureImplemented(FEAT_SSBS2) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    PSTATE.SSBS = X[t, 64]<12>;
elsif PSTATE.EL == EL1 then
    PSTATE.SSBS = X[t, 64]<12>;
elsif PSTATE.EL == EL2 then
    PSTATE.SSBS = X[t, 64]<12>;
elsif PSTATE.EL == EL3 then
    PSTATE.SSBS = X[t, 64]<12>;
```

```
MSR SSBS, #<imm>
```

op0	op1	CRn	op2
0b00	0b011	0b0100	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.27 SVCR, Streaming Vector Control Register

The SVCR characteristics are:

Purpose

Allows access to the Streaming SVE mode in PSTATE.SM, and the SME storage enable in PSTATE.ZA.

Configuration

This register is present only when FEAT_SME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SVCR are UNDEFINED.

Attributes

SVCR is a 64-bit register.

Field descriptions

Bits [63:2]

Reserved, RES0.

ZA, bit [1]

Access to PSTATE.ZA, SME storage enable.

When this storage is disabled, execution of an instruction which can access it is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of {`0x1D`, `0x3`}.

The possible values of this bit are:

ZA	Meaning
`0b0`	SME ZA storage and, if implemented, ZT0 storage are invalid and not accessible. This control causes execution at any Exception level of instructions that can access this storage to be trapped.
`0b1`	SME ZA storage and, if implemented, ZT0 storage are valid and accessible. This control does not cause execution of any instructions to be trapped.

When a write to SVCR.ZA changes the value of PSTATE.ZA from 0 to 1, all implemented bits of the storage are set to zero.

Changes to this field do not have an effect on the SVE vector and predicate registers and FPSR.

A direct or indirect read of ZA appears to occur in program order relative to a direct write of SVCR, and to `MSR SVCRZA` and `MSR SVCRSMZA` instructions, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, PSTATE.ZA resets to ‘0’.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SM, bit [0]

Access to PSTATE.SM, Streaming SVE mode.

When the PE is in Streaming SVE mode, the Streaming SVE vector length (SVL) applies to SVE instructions, and execution at any Exception level of an instruction which is illegal in that mode is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of {`0x1D`, `0x1`}.

When the PE is not in Streaming SVE mode, the SVE vector length (VL) applies to SVE instructions, and execution at any Exception level of an instruction which is only legal in that mode is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of {`0x1D`, `0x2`}.

The possible values of this bit are:

SM	Meaning
`0b0`	The PE is not in Streaming SVE mode.
`0b1`	The PE is in Streaming SVE mode.

When a write to SVCR.SM changes the value of PSTATE.SM, the following applies:

- When changed from 0 to 1, an entry to Streaming SVE mode is performed.
- When changed from 1 to 0, an exit from Streaming SVE mode is performed.
- All implemented bits of the SVE registers Z0-Z31, P0-P15, and FFR in the new mode are set to zero.
- All bits in FPMR are set to zero.
- FPSR in the new mode is set to `0x0000`_0000_0800_009f, in which all cumulative status bits are set to 1.

Changes to this field do not have an effect on SME ZA storage or, if implemented, ZT0 storage.

A direct or indirect read of SM appears to occur in program order relative to a direct write of SVCR, and to `MSR SVCRSM` and `MSR SVCRSMZA` instructions, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, PSTATE.SM resets to ‘0’.

Accessing SVCR

SVCR is read/write and can be accessed from any Exception level.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SVCR
```

op0	op1	CRn	CRm	op2
`0b11`	`0b011`	`0b0100`	`0b0010`	`0b010`

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif !ELIsInHost(EL0) && CPACR_EL1.SMEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        else
            AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif ELIsInHost(EL0) && CPTR_EL2.SMEN != '11' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2.SMEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SVCR;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif CPACR_EL1.SMEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2.SMEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SVCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif !ELIsInHost(EL2) && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2.SMEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SVCR;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SVCR;
```

```
MSR SVCR, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif !ELIsInHost(EL0) && CPACR_EL1.SMEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64.SystemAccessTrap(EL2, 0x1D);
            else
                AArch64.SystemAccessTrap(EL1, 0x1D);
        elsif ELIsInHost(EL0) && CPTR_EL2.SMEN != '11' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif ELIsInHost(EL2) && CPTR_EL2.SMEN IN {'x0'} then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TSM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            SVCR = X[t, 64];
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif CPACR_EL1.SMEN IN {'x0'} then
            AArch64.SystemAccessTrap(EL1, 0x1D);
        elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2.TSM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif ELIsInHost(EL2) && CPTR_EL2.SMEN IN {'x0'} then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            SVCR = X[t, 64];
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif !ELIsInHost(EL2) && CPTR_EL2.TSM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif ELIsInHost(EL2) && CPTR_EL2.SMEN IN {'x0'} then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            SVCR = X[t, 64];
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.ESM == '0' then
            AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            SVCR = X[t, 64];
```

```
MSR SVCRSM, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b001x	0b011

```
MSR SVCRZA, #<imm>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b010x	0b011

```
MSR SVCRSMZA, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b011x	0b011

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.28 TCO, Tag Check Override

The TCO characteristics are:

Purpose

When FEAT_MTE is implemented, this register allows tag checks to be disabled globally.

When FEAT_MTE2 is not implemented, it is IMPLEMENTATION DEFINED if accesses to this register access PSTATE.TCO or are RAZ/WI.

Configuration

This register is present only when FEAT_MTE is implemented. Otherwise, direct accesses to TCO are UNDEFINED.

Attributes

TCO is a 64-bit register.

Field descriptions

Bits [63:26]

Reserved, RES0.

TCO, bit [25]

Allows memory tag checks to be globally disabled.

TCO	Meaning
0b0	Loads and Stores are not affected by this control.
0b1	Loads and Stores are unchecked.

Bits [24:0]

Reserved, RES0.

Accessing TCO

For information about the operation of the MSR (immediate) accessor, see MSR (immediate).

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCO
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b111

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    X[t, 64] = Zeros(38):PSTATE.TCO:Zeros(25);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(38):PSTATE.TCO:Zeros(25);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(38):PSTATE.TCO:Zeros(25);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(38):PSTATE.TCO:Zeros(25);
```

```
MSR TCO, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b111

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    PSTATE.TCO = X[t, 64]<25>;
elsif PSTATE.EL == EL1 then
    PSTATE.TCO = X[t, 64]<25>;
elsif PSTATE.EL == EL2 then
    PSTATE.TCO = X[t, 64]<25>;
elsif PSTATE.EL == EL3 then
    PSTATE.TCO = X[t, 64]<25>;
```

```
MSR TCO, #<imm>
```

op0	op1	CRn	op2
0b00	0b011	0b0100	0b100

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.29 UAO, User Access Override

The UAO characteristics are:

Purpose

Allows access to the User Access Override bit.

Configuration

This register is present only when FEAT_UAO is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to UAO are UNDEFINED.

Attributes

UAO is a 64-bit register.

Field descriptions

Bits [63:24]

Reserved, RES0.

UAO, bit [23]

User Access Override.

UAO	Meaning
0b0	Explicit Memory Effects generated by unprivileged memory access instructions and described as unprivileged are unprivileged.
0b1	Explicit Memory Effects generated by unprivileged memory access instructions and described as unprivileged are privileged.

The Effective value of PSTATE.UAO is 0 when execution is at EL0.

The Effective value of PSTATE.UAO is 1 when any of the following is true:

- Execution is at EL1, and HCR_EL2.{NV, NV1} is {1, 1}.
- Execution is at EL2, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- Execution is at EL3.

Bits [22:0]

Reserved, RES0.

Accessing UAO

For more information about the operation of the MSR (immediate) accessor, see ‘MSR (immediate)’.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, UAO
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_UAO) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(40):PSTATE.UAO:Zeros(23);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(40):PSTATE.UAO:Zeros(23);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(40):PSTATE.UAO:Zeros(23);
```

```
MSR UAO, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_UAO) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.UAO = X[t, 64]<23>;
elsif PSTATE.EL == EL2 then
    PSTATE.UAO = X[t, 64]<23>;
elsif PSTATE.EL == EL3 then
    PSTATE.UAO = X[t, 64]<23>;
```

```
MSR UAO, #<imm>
```

op0	op1	CRn	op2
0b00	0b000	0b0100	0b011

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3 A64 System instructions for cache maintenance

This section lists the A64 System instructions for cache maintenance.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.1 DC CGDSW, Clean of Data and Allocation Tags by Set/Way

The DC CGDSW characteristics are:

Purpose

Clean data and Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CGDSW are UNDEFINED.

Attributes

DC CGDSW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CGDSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b110

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.2 DC CGDVAC, Clean of Data and Allocation Tags by VA to PoC

The DC CGDVAC characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVAC are UNDEFINED.

Attributes

DC CGDVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b101

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) &&
        ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.3 DC CGDVADP, Clean of Data and Allocation Tags by VA to PoDP

The DC CGDVADP characteristics are:

Purpose

Clean Allocation Tags and data in data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CGDVAP.

Configuration

This system instruction is present only when FEAT_DPB2 is implemented and FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVADP are UNDEFINED.

Attributes

DC CGDVADP is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVADP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDVADP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b101

```
if !(IsFeatureImplemented(FEAT_DPB2) && IsFeatureImplemented(FEAT_MTE)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) &&
        ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.4 DC CGDVAOC, Clean of Data and Allocation Tags by VA to Outer Cache level

The DC CGDVAOC characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Outer cache level.

Configuration

This system instruction is present only when FEAT_OCCMO is implemented, FEAT_MTE is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CGDVAOC are UNDEFINED.

Attributes

DC CGDVAOC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVAOC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDVAOC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1011	0b111

```
if !(IsFeatureImplemented(FEAT_OCCMO) && IsFeatureImplemented(FEAT_MTE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_OuterCache);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.5 DC CGDVAP, Clean of Data and Allocation Tags by VA to PoP

The DC CGDVAP characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CGDVAC.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVAP are UNDEFINED.

Attributes

DC CGDVAP is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVAP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDVAP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b101

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.6 DC CGSW, Clean of Allocation Tags by Set/Way

The DC CGSW characteristics are:

Purpose

Clean Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CGSW are UNDEFINED.

Attributes

DC CGSW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CGSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b100

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.7 DC CGVAC, Clean of Allocation Tags by VA to PoC

The DC CGVAC characteristics are:

Purpose

Clean Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVAC are UNDEFINED.

Attributes

DC CGVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) &&
        ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.8 DC CGVADP, Clean of Allocation Tags by VA to PoDP

The DC CGVADP characteristics are:

Purpose

Clean Allocation tags by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CGVAP.

Configuration

This system instruction is present only when FEAT_DPB2 is implemented and FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVADP are UNDEFINED.

Attributes

DC CGVADP is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGVADP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGVADP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_DPB2) && IsFeatureImplemented(FEAT_MTE)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) &&
        ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.9 DC CGVAP, Clean of Allocation Tags by VA to PoP

The DC CGVAP characteristics are:

Purpose

Clean Allocation Tags in data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CGVAC.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVAP are UNDEFINED.

Attributes

DC CGVAP is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGVAP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGVAP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b011

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.10 DC CIGDPAE, Clean and invalidate of data and allocation tags by PA to PoE

The DC CIGDPAE characteristics are:

Purpose

Clean and invalidate of data and allocation tags by PA to PoE.

Configuration

This system instruction is present only when FEAT_MEC is implemented, FEAT_MTE2 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIGDPAE are UNDEFINED.

Attributes

DC CIGDPAE is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with the NSE2 and NSE field, this field specifies the target physical address space.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	Reserved.
0b0	0b0	0b1	Reserved.
0b0	0b1	0b0	Reserved.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

If {NSE2, NSE, NS} is reserved, then no cache entries are required to be cleaned or invalidated.

Otherwise:

Together with the NSE field, this field specifies the target physical address space.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Meaning
0b0	0b0	Reserved.
0b0	0b1	Reserved.
0b1	0b0	Reserved.
0b1	0b1	Realm.

If {NSE, NS} is not {1, 1}, then no cache entries are required to be cleaned or invalidated.

NSE, bit [62]

If FEAT_RME_GDI is implemented, this field together with the NS and NSE2 fields, specifies the target physical address space.

Otherwise, this field and the NS field specify the physical address space

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see DC CIGDPAE.NS.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with the NS and NSE field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIGDPAE.NS.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to PA[51:0] if ID_AA64MMFR0_EL1.PARange = 0111. For more information, see PA[51:0].

Otherwise:

Reserved, RES0.

PA, bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

Executing DC CIGDPAE

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDPAE, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1110	0b111

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_MTE2) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        UNDEFINED;
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoE);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoE);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.11 DC CIGDPAPA, Clean and Invalidate of Data and Allocation Tags by PA to PoPA

The DC CIGDPAPA characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by physical address to the Point of Physical Aliasing.

Note

This instruction cleans and invalidates all copies of the Location specified in the Xt argument, irrespective of any MECID associated with the Location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.

Configuration

This system instruction is present only when FEAT_RME is implemented, FEAT_MTE2 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIGDPAPA are UNDEFINED.

Attributes

DC CIGDPAPA is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with the NSE2 and NSE field, this field specifies the target physical address space.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b0	0b1	Non-secure.
0b0	0b1	0b0	Root.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

If {NSE2, NSE, NS} is reserved, then no cache entries are required to be cleaned or invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Together with the NSE field, this field specifies the target physical address space.

NSE	NS	Meaning
`0b0`	`0b0`	When Secure state is implemented, Secure. Otherwise reserved.
`0b0`	`0b1`	Non-secure.
`0b1`	`0b0`	Root.
`0b1`	`0b1`	Realm.

If FEAT_SEL2 is not implemented, and {NSE, NS} == {`0b0`, `0b0`}, then no cache entries are required to be cleaned or invalidated

NSE, bit [62]

If FEAT_RME_GDI is implemented, this field together with the NS and NSE2 fields, specifies the target physical address space.

Otherwise, this field and the NS field specify the physical address space

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see DC CIGDPAPA.NS.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with the NS and NSE field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIGDPAPA.NS.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to PA[51:0] if ID_AA64MMFR0_EL1.PARange = 0111. For more information see PA[51:0].

Otherwise:

Reserved, RES0.

PA, bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

Executing DC CIGDPAPA

- This instruction is not subject to any translation, permission checks, or granule protection checks.
- This instruction affects all caches in the Outer Shareable shareability domain.
- This instruction has the same ordering, observability, and completion behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
DC CIGDPAPA, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1110	0b101

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_MTE2) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoPA);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.12 DC CIGDSW, Clean and Invalidate of Data and Allocation Tags by Set/Way

The DC CIGDSW characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CIGDSW are UNDEFINED.

Attributes

DC CIGDSW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CIGDSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b110

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.13 DC CIGDVAC, Clean and Invalidate of Data and Allocation Tags by VA to PoC

The DC CIGDVAC characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CIGDVAC are UNDEFINED.

Attributes

DC CIGDVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGDVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b101

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
        ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.14 DC CIGDVAOC, Clean and Invalidate of Data and Allocation Tags by VA to Outer Cache level

The DC CIGDVAOC characteristics are:

Purpose

Clean and invalidate of data and Allocation Tags in data cache by address to Outer cache level.

Configuration

This system instruction is present only when FEAT_OCCMO is implemented, FEAT_MTE is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIGDVAOC are UNDEFINED.

Attributes

DC CIGDVAOC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGDVAOC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDVAOC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1111	0b111

```
if !(IsFeatureImplemented(FEAT_OCCMO) && IsFeatureImplemented(FEAT_MTE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.15 DC CIGDVAPS, Clean and Invalidate of Data and Allocation Tags by VA to PoPS

The DC CIGDVAPS characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by virtual address to the Point of Physical Storage.

Note

This instruction cleans and invalidates all copies of the Location specified in the Xt argument, irrespective of any MECID associated with the Location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.

Configuration

This system instruction is present only when FEAT_PoPS is implemented, FEAT_MTE2 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIGDVAPS are UNDEFINED.

Attributes

DC CIGDVAPS is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGDVAPS

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDVAPS, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1111	0b101

```
if !(IsFeatureImplemented(FEAT_PoPS) && IsFeatureImplemented(FEAT_MTE2) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPCP == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3.FGTEn2 == '0') ||
         ↪HFGITR2_EL2.nDCCIVAPS == '0') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.16 DC CIGSW, Clean and Invalidate of Allocation Tags by Set/Way

The DC CIGSW characteristics are:

Purpose

Clean and Invalidate Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CIGSW are UNDEFINED.

Attributes

DC CIGSW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CIGSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b100

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.17 DC CIGVAC, Clean and Invalidate of Allocation Tags by VA to PoC

The DC CIGVAC characteristics are:

Purpose

Clean and Invalidate Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CIGVAC are UNDEFINED.

Attributes

DC CIGVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b011

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
        ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.18 DC CIPAE, Data or unified Cache line Clean and Invalidate by PA to PoE

The DC CIPAE characteristics are:

Purpose

Data or unified Cache line Clean and Invalidate by PA to PoE.

Configuration

This system instruction is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIPAE are UNDEFINED.

Attributes

DC CIPAE is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with the NSE2 and NSE field, this field specifies the target physical address space.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	Reserved.
0b0	0b0	0b1	Reserved.
0b0	0b1	0b0	Reserved.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

If {NSE2, NSE, NS} is reserved, then no cache entries are required to be cleaned or invalidated.

Otherwise:

Together with the NSE field, this field specifies the target physical address space.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Meaning
0b0	0b0	Reserved.
0b0	0b1	Reserved.
0b1	0b0	Reserved.
0b1	0b1	Realm.

If {NSE, NS} is not {1, 1}, then no cache entries are required to be cleaned or invalidated.

NSE, bit [62]

If FEAT_RME_GDI is implemented, this field together with the NS and NSE2 fields, specifies the target physical address space.

Otherwise, this field and the NS field specify the physical address space

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see DC CIPAE.NS.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with the NS and NSE field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIPAE.NS.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to PA[51:0] if ID_AA64MMFR0_EL1.PARange = 0111. For more information see PA[51:0].

Otherwise:

Reserved, RES0.

PA, bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

Executing DC CIPAE

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIPAE, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1110	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        UNDEFINED;
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoE);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoE);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.19 DC CIPAPA, Data or unified Cache line Clean and Invalidate by PA to PoPA

The DC CIPAPA characteristics are:

Purpose

Clean and Invalidate data cache by physical address to the Point of Physical Aliasing.

Note

This instruction cleans and invalidates all copies of the Location specified in the Xt argument, irrespective of any MECID associated with the Location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIPAPA are UNDEFINED.

Attributes

DC CIPAPA is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with the NSE2 and NSE field, this field specifies the target physical address space.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b0	0b1	Non-secure.
0b0	0b1	0b0	Root.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

If {NSE2, NSE, NS} is reserved, then no cache entries are required to be cleaned or invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Together with the NSE field, this field specifies the target physical address space.

NSE	NS	Meaning
`0b0`	`0b0`	When Secure state is implemented, Secure. Otherwise reserved.
`0b0`	`0b1`	Non-secure.
`0b1`	`0b0`	Root.
`0b1`	`0b1`	Realm.

If FEAT_SEL2 is not implemented, and {NSE, NS} == {`0b0`, `0b0`}, then no cache entries are required to be cleaned or invalidated

NSE, bit [62]

If FEAT_RME_GDI is implemented, this field together with the NS and NSE2 fields, specifies the target physical address space.

Otherwise, this field and the NS field specify the physical address space

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see DC CIPAPA.NS.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with the NS and NSE field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIPAPA.NS.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to PA[51:0] if ID_AA64MMFR0_EL1.PARange = 0111. For more information see PA[51:0].

Otherwise:

Reserved, RES0.

PA, bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

Executing DC CIPAPA

- This instruction is not subject to any translation, permission checks, or granule protection checks.
- This instruction affects all caches in the Outer Shareable shareability domain.
- This instruction has the same ordering, observability, and completion behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
DC CIPAPA, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1110	0b001

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoPA);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.20 DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way

The DC CISW characteristics are:

Purpose

Clean and Invalidate data cache by set/way.

Configuration

AArch64 System instruction DC CISW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CISW are UNDEFINED.

Attributes

DC CISW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CISW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.21 DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC

The DC CIVAC characteristics are:

Purpose

Clean and Invalidate data cache by address to Point of Coherency.

Configuration

AArch64 System instruction DC CIVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIVAC are UNDEFINED.

Attributes

DC CIVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
        ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.22 DC CIVAOC, Data or unified Cache line Clean and Invalidate by VA to Outer Cache level

The DC CIVAOC characteristics are:

Purpose

Clean and Invalidate data cache by address to Outer cache level.

Configuration

This system instruction is present only when FEAT_OCCMO is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIVAOC are UNDEFINED.

Attributes

DC CIVAOC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIVAOC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIVAOC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1111	0b000

```
if !(IsFeatureImplemented(FEAT_OCCMO) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.23 DC CIVAPS, Clean and Invalidate of Data by VA to PoPS

The DC CIVAPS characteristics are:

Purpose

Clean and Invalidate data in data cache by virtual address to the Point of Physical Storage.

Note

This instruction cleans and invalidates all copies of the Location specified in the Xt argument, irrespective of any MECID associated with the Location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.

Configuration

This system instruction is present only when FEAT_PoPS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIVAPS are UNDEFINED.

Attributes

DC CIVAPS is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIVAPS

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIVAPS, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1111	0b001

```
if !(IsFeatureImplemented(FEAT_PoPS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3.FGTEn2 == '0') ||
        ↪HFGITR2_EL2.nDCCIVAPS == '0') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.24 DC CSW, Data or unified Cache line Clean by Set/Way

The DC CSW characteristics are:

Purpose

Clean data cache by set/way.

Configuration

AArch64 System instruction DC CSW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CSW are UNDEFINED.

Attributes

DC CSW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.25 DC CVAC, Data or unified Cache line Clean by VA to PoC

The DC CVAC characteristics are:

Purpose

Clean data cache by address to Point of Coherency.

Configuration

AArch64 System instruction DC CVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVAC are UNDEFINED.

Attributes

DC CVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.26 DC CVADP, Data or unified Cache line Clean by VA to PoDP

The DC CVADP characteristics are:

Purpose

Clean data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CVAP.

Configuration

This system instruction is present only when FEAT_DPB2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVADP are UNDEFINED.

Attributes

DC CVADP is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVADP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVADP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b001

```
if !(IsFeatureImplemented(FEAT_DPB2) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) &&
        ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.27 DC CVAOC, Data or unified Cache line Clean by VA to Outer Cache level

The DC CVAOC characteristics are:

Purpose

Clean data cache by address to Outer cache level.

Configuration

This system instruction is present only when FEAT_OCCMO is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVAOC are UNDEFINED.

Attributes

DC CVAOC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAOC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVAOC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1011	0b000

```
if !(IsFeatureImplemented(FEAT_OCCMO) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_OuterCache);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.28 DC CVAP, Data or unified Cache line Clean by VA to PoP

The DC CVAP characteristics are:

Purpose

Clean data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CVAC.

Configuration

This system instruction is present only when FEAT_DPB is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVAP are UNDEFINED.

Attributes

DC CVAP is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVAP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b001

```
if !(IsFeatureImplemented(FEAT_DPB) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) &&
        ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.29 DC CVAU, Data or unified Cache line Clean by VA to PoU

The DC CVAU characteristics are:

Purpose

Clean data cache by address to Point of Unification.

Configuration

AArch64 System instruction DC CVAU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVAU are UNDEFINED.

Attributes

DC CVAU is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAU

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVAU, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCCVAU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.30 DC GVA, Data Cache set Allocation Tag by VA

The DC GVA characteristics are:

Purpose

Write a value to the Allocation Tags of a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0. The Allocation Tag used is determined by the input address.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC GVA are UNDEFINED.

Attributes

DC GVA is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing DC GVA

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

A DC GVA instruction to any type of Device memory is CONSTRAINED UNPREDICTABLE between:

- Not modifying Allocation tags at the specified addresses.
- Generating an Alignment Fault determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of stores to each Allocation Tag within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC GVA, <Xt>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b011

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.DZE == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.DZE == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.MemZero(X[t, 64], CacheType_Tag);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.MemZero(X[t, 64], CacheType_Tag);
elsif PSTATE.EL == EL2 then
    AArch64.MemZero(X[t, 64], CacheType_Tag);
elsif PSTATE.EL == EL3 then
    AArch64.MemZero(X[t, 64], CacheType_Tag);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.31 DC GZVA, Data Cache set Allocation Tags and Zero by VA

The DC GZVA characteristics are:

Purpose

Zero data and write a value to the Allocation Tags of a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0. The Allocation Tag used is determined by the input address.

This instruction might have the poison-atomic property for IMPLEMENTATION DEFINED regions of physical memory.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC GZVA are UNDEFINED.

Attributes

DC GZVA is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing DC GZVA

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

If the memory region being zeroed is any type of Device memory, this instruction generates an alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte and Allocation tag within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC GZVA, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b100

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_MTE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.DZE == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.DZE == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.MemZero(X[t, 64], CacheType_Data_Tag);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.MemZero(X[t, 64], CacheType_Data_Tag);
elsif PSTATE.EL == EL2 then
    AArch64.MemZero(X[t, 64], CacheType_Data_Tag);
elsif PSTATE.EL == EL3 then
    AArch64.MemZero(X[t, 64], CacheType_Data_Tag);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.32 DC IGDSW, Invalidate of Data and Allocation Tags by Set/Way

The DC IGDSW characteristics are:

Purpose

Invalidate data and Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGDSW are UNDEFINED.

Attributes

DC IGDSW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC IGDSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- No cache lines.
- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IGDSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b110

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.33 DC IGDVAC, Invalidate of Data and Allocation Tags by VA to PoC

The DC IGDVAC characteristics are:

Purpose

Invalidate data and Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGDVAC are UNDEFINED.

Attributes

DC IGDVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC IGDVAC

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IGDVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b101

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) &&
        ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.34 DC IGSW, Invalidate of Allocation Tags by Set/Way

The DC IGSW characteristics are:

Purpose

Invalidate Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGSW are UNDEFINED.

Attributes

DC IGSW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC IGSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- No cache lines.
- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IGSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b100

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.35 DC IGVAC, Invalidate of Allocation Tags by VA to PoC

The DC IGVAC characteristics are:

Purpose

Invalidate Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGVAC are UNDEFINED.

Attributes

DC IGVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC IGVAC

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IGVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b011

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) &&
        ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.36 DC ISW, Data or unified Cache line Invalidate by Set/Way

The DC ISW characteristics are:

Purpose

Invalidate data cache by set/way.

When FEAT_MTE2 is implemented, this instruction might invalidate Allocation Tags from caches. When it invalidates Allocation Tags from caches, it also cleans them.

Configuration

AArch64 System instruction DC ISW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC ISW are UNDEFINED.

Attributes

DC ISW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing DC ISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC ISW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.37 DC IVAC, Data or unified Cache line Invalidate by VA to PoC

The DC IVAC characteristics are:

Purpose

Invalidate data cache by address to Point of Coherency.

When FEAT_MTE2 is implemented, this instruction might invalidate Allocation Tags from caches. When it invalidates Allocation Tags from caches, it also cleans them.

Configuration

AArch64 System instruction DC IVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC IVAC are UNDEFINED.

Attributes

DC IVAC is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC IVAC

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) &&
         ↪!AArch64.CanTrapDC(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.38 DC ZVA, Data Cache Zero by VA

The DC ZVA characteristics are:

Purpose

Zero data cache by address. Zeroes a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0.

This instruction might have the poison-atomic property for IMPLEMENTATION DEFINED regions of physical memory.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC ZVA are UNDEFINED.

Attributes

DC ZVA is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing DC ZVA

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 0.

If the memory region being zeroed is any type of Device memory, this instruction generates an Alignment fault which is prioritized in the same way as other Alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC ZVA, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.DZE == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.DZE == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.MemZero(X[t, 64], CacheType_Data);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.MemZero(X[t, 64], CacheType_Data);
elsif PSTATE.EL == EL2 then
    AArch64.MemZero(X[t, 64], CacheType_Data);
elsif PSTATE.EL == EL3 then
    AArch64.MemZero(X[t, 64], CacheType_Data);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.39 IC IALLU, Instruction Cache Invalidate All to PoU

The IC IALLU characteristics are:

Purpose

Invalidate all instruction caches of the PE executing the instruction to the Point of Unification.

Configuration

AArch64 System instruction IC IALLU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to IC IALLU are UNDEFINED.

Attributes

IC IALLU is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing IC IALLU

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
IC IALLU{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) &&
        ↪!AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪HFGITR_EL2.ICIALLU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLUIS);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLU);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) &&
     ↪!AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLU);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch64.IC(CacheOpScope_ALLU);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.40 IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The IC IALLUIS characteristics are:

Purpose

Invalidate all instruction caches in the Inner Shareable domain of the PE executing the instruction to the Point of Unification.

Configuration

AArch64 System instruction IC IALLUIS bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to IC IALLUIS are UNDEFINED.

Attributes

IC IALLUIS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing IC IALLUIS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
IC IALLUIS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) &&
        ↪!AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TICAB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪HFGITR_EL2.ICIALLUIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.IC(CacheOpScope_ALLUIS);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) &&
         ↪!AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLUIS);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch64.IC(CacheOpScope_ALLUIS);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.41 IC IVAU, Instruction Cache line Invalidate by VA to PoU

The IC IVAU characteristics are:

Purpose

Invalidate instruction cache by address to Point of Unification.

Configuration

AArch64 System instruction IC IVAU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to IC IVAU are UNDEFINED.

Attributes

IC IVAU is a 64-bit System instruction.

Field descriptions

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing IC IVAU

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The instruction cache maintenance instruction (IC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
IC IVAU{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) &&
         ↪!AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.ICIVAU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(X[t, 64], CacheOpScope_PoU);
elsif PSTATE.EL == EL1 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) &&
         ↪!AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.ICIVAU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(X[t, 64], CacheOpScope_PoU);
elsif PSTATE.EL == EL2 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) &&
         ↪!AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(X[t, 64], CacheOpScope_PoU);
elsif PSTATE.EL == EL3 then
    if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch64.IC(X[t, 64], CacheOpScope_PoU);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4 A64 System instructions for address translation

This section lists the A64 System instructions for address translation.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.1 AT S12E0R, Address Translate Stages 1 and 2 EL0 Read

The AT S12E0R characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if reading from the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S12E0R are UNDEFINED.

Attributes

AT S12E0R is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S12E0R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S12E0R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b110

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) || HCR_EL2.<DC,VM> == '00' then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
    elsif EL2Enabled() && (ELIsInHost(EL0) || HCR_EL2.<DC,VM> == '00') then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Read);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.2 AT S12E0W, Address Translate Stages 1 and 2 EL0 Write

The AT S12E0W characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if writing to the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S12E0W are UNDEFINED.

Attributes

AT S12E0W is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S12E0W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S12E0W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b111

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) || HCR_EL2.<DC,VM> == '00' then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
    elsif EL2Enabled() && (ELIsInHost(EL0) || HCR_EL2.<DC,VM> == '00') then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Write);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.3 AT S12E1R, Address Translate Stages 1 and 2 EL1 Read

The AT S12E1R characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if reading from the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S12E1R are UNDEFINED.

Attributes

AT S12E1R is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S12E1R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S12E1R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b100

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) || HCR_EL2.<DC,VM> == '00' then
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
    elsif EL2Enabled() && (ELIsInHost(EL0) || HCR_EL2.<DC,VM> == '00') then
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Read);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.4 AT S12E1W, Address Translate Stages 1 and 2 EL1 Write

The AT S12E1W characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if writing to the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S12E1W are UNDEFINED.

Attributes

AT S12E1W is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S12E1W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S12E1W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b101

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) || HCR_EL2.<DC,VM> == '00' then
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
    elsif EL2Enabled() && (ELIsInHost(EL0) || HCR_EL2.<DC,VM> == '00') then
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Write);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.5 AT S1E0R, Address Translate Stage 1 EL0 Read

The AT S1E0R characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if reading from the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E0R are UNDEFINED.

Attributes

AT S1E0R is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E0R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E0R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b010

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.ATS1E0R == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.6 AT S1E0W, Address Translate Stage 1 EL0 Write

The AT S1E0W characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if writing to the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E0W are UNDEFINED.

Attributes

AT S1E0W is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E0W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E0W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b011

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.ATS1E0W == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.7 AT S1E1A, Address Translate Stage 1 EL1 Without Permission checks

The AT S1E1A characteristics are:

Purpose

Performs a stage 1 address translation, while ignoring the permission checks using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_ATS1A is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1A are UNDEFINED.

Attributes

AT S1E1A is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E1A

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1A, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b010

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ATS1A) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.ATS1E1A == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Any);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Any);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Any);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.8 AT S1E1R, Address Translate Stage 1 EL1 Read

The AT S1E1R characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if reading from the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1R are UNDEFINED.

Attributes

AT S1E1R is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E1R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.ATS1E1R == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.9 AT S1E1RP, Address Translate Stage 1 EL1 Read PAN

The AT S1E1RP characteristics are:

Purpose

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a read from a location will generate a Permission fault for a privileged access, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_PAN2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1RP are UNDEFINED.

Attributes

AT S1E1RP is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E1RP

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1RP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_PAN2) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.ATS1E1RP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_ReadPAN);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_ReadPAN);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_ReadPAN);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.10 AT S1E1W, Address Translate Stage 1 EL1 Write

The AT S1E1W characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if writing to the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1W are UNDEFINED.

Attributes

AT S1E1W is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E1W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.ATS1E1W == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.11 AT S1E1WP, Address Translate Stage 1 EL1 Write PAN

The AT S1E1WP characteristics are:

Purpose

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a write to a location will generate a Permission fault for a privileged access, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_PAN2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1WP are UNDEFINED.

Attributes

AT S1E1WP is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E1WP

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1WP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_PAN2) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.ATS1E1WP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_WritePAN);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_WritePAN);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_WritePAN);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.12 AT S1E2A, Address Translate Stage 1 EL2 Without Permission checks

The AT S1E2A characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, while ignoring permissions checks from the given virtual address.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_ATS1A is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E2A are UNDEFINED.

Attributes

AT S1E2A is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E2A

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E2A, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1001	0b010

```
if !(IsFeatureImplemented(FEAT_ATS1A) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Any);
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !EL2Enabled() then
    UNDEFINED;
else
    AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Any);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.13 AT S1E2R, Address Translate Stage 1 EL2 Read

The AT S1E2R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, with permissions as if reading from the given virtual address.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E2R are UNDEFINED.

Attributes

AT S1E2R is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E2R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E2R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Read);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.14 AT S1E2W, Address Translate Stage 1 EL2 Write

The AT S1E2W characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, with permissions as if writing to the given virtual address.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E2W are UNDEFINED.

Attributes

AT S1E2W is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E2W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E2W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Write);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.15 AT S1E3A, Address Translate Stage 1 EL3 Without Permission checks

The AT S1E3A characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, while ignoring permissions checks from the given virtual address.

Configuration

This system instruction is present only when FEAT_ATS1A is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E3A are UNDEFINED.

Attributes

AT S1E3A is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E3A

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E3A, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1001	0b010

```
if !(IsFeatureImplemented(FEAT_ATS1A) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL3, ATAccess_Any);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.16 AT S1E3R, Address Translate Stage 1 EL3 Read

The AT S1E3R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if reading from the given virtual address.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E3R are UNDEFINED.

Attributes

AT S1E3R is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E3R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E3R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL3, ATAccess_Read);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.17 AT S1E3W, Address Translate Stage 1 EL3 Write

The AT S1E3W characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if writing to the given virtual address.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E3W are UNDEFINED.

Attributes

AT S1E3W is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E3W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E3W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL3, ATAccess_Write);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5 A64 System instructions for TLB maintenance

This section lists the A64 System instructions for TLB maintenance.

For more information about these instructions see TLB maintenance instructions. In particular, for the full description of the scope of each instruction see TLB maintenance instruction scope.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.1 TLBI ALLE1, TLBI ALLE1NXS, TLB Invalidate All, EL1

The TLBI ALLE1, TLBI ALLE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation only applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE1, TLBI ALLE1NXS are UNDEFINED.

Attributes

TLBI ALLE1, TLBI ALLE1NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1, TLBI ALLE1NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI ALLE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.2 TLBI ALLE1IS, TLBI ALLE1ISNXS, TLB Invalidate All, EL1, Inner Shareable

The TLBI ALLE1IS, TLBI ALLE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE1IS, TLBI ALLE1ISNXS are UNDEFINED.

Attributes

TLBI ALLE1IS, TLBI ALLE1ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1IS, TLBI ALLE1ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI ALLE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.3 TLBI ALLE1OS, TLBI ALLE1OSNXS, TLB Invalidate All, EL1, Outer Shareable

The TLBI ALLE1OS, TLBI ALLE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE1OS, TLBI ALLE1OSNXS are UNDEFINED.

Attributes

TLBI ALLE1OS, TLBI ALLE1OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1OS, TLBI ALLE1OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI ALLE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.4 TLBI ALLE2, TLBI ALLE2NXS, TLB Invalidate All, EL2

The TLBI ALLE2, TLBI ALLE2NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation only applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE2, TLBI ALLE2NXS are UNDEFINED.

Attributes

TLBI ALLE2, TLBI ALLE2NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2, TLBI ALLE2NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI ALLE2NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_NSH, TLBI_ExcludeXS, X[t,
            ↪64]);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_NSH, TLBI_ExcludeXS,
                 ↪X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_ExcludeXS,
                 ↪X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.5 TLBI ALLE2IS, TLBI ALLE2ISNXS, TLB Invalidate All, EL2, Inner Shareable

The TLBI ALLE2IS, TLBI ALLE2ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE2IS, TLBI ALLE2ISNXS are UNDEFINED.

Attributes

TLBI ALLE2IS, TLBI ALLE2ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2IS, TLBI ALLE2ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI ALLE2ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_ISH, TLBI_ExcludeXS, X[t,
            ↪64]);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_ISH, TLBI_ExcludeXS,
                 ↪X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_ExcludeXS,
                 ↪X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.6 TLBI ALLE2OS, TLBI ALLE2OSNXS, TLB Invalidate All, EL2, Outer Shareable

The TLBI ALLE2OS, TLBI ALLE2OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE2OS, TLBI ALLE2OSNXS are UNDEFINED.

Attributes

TLBI ALLE2OS, TLBI ALLE2OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2OS, TLBI ALLE2OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI ALLE2OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_OSH, TLBI_ExcludeXS, X[t,
            ↪64]);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Broadcast_OSH, TLBI_ExcludeXS,
                 ↪X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_ExcludeXS,
                 ↪X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.7 TLBI ALLE3, TLBI ALLE3NXS, TLB Invalidate All, EL3

The TLBI ALLE3, TLBI ALLE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE3, TLBI ALLE3NXS are UNDEFINED.

Attributes

TLBI ALLE3, TLBI ALLE3NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3, TLBI ALLE3NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI ALLE3NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_NSH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.8 TLBI ALLE3IS, TLBI ALLE3ISNXS, TLB Invalidate All, EL3, Inner Shareable

The TLBI ALLE3IS, TLBI ALLE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE3IS, TLBI ALLE3ISNXS are UNDEFINED.

Attributes

TLBI ALLE3IS, TLBI ALLE3ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3IS, TLBI ALLE3ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI ALLE3ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_ISH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.9 TLBI ALLE3OS, TLBI ALLE3OSNXS, TLB Invalidate All, EL3, Outer Shareable

The TLBI ALLE3OS, TLBI ALLE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE3OS, TLBI ALLE3OSNXS are UNDEFINED.

Attributes

TLBI ALLE3OS, TLBI ALLE3OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3OS, TLBI ALLE3OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b000

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI ALLE3OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_OSH, TLBI_ExcludeXS, X[t,
            ↪64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.10 TLBI ASIDE1, TLBI ASIDE1NXS, TLB Invalidate by ASID, EL1

The TLBI ASIDE1, TLBI ASIDE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ASIDE1, TLBI ASIDE1NXS are UNDEFINED.

Attributes

TLBI ASIDE1, TLBI ASIDE1NXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1, TLBI ASIDE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ASIDE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIASIDE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr,
                 ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBI_AllAttr,
             ↪X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr, X[t,
             ↪64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr,
                 ↪X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI ASIDE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIASIDE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
             ↪TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.11 TLBI ASIDE1IS, TLBI ASIDE1ISNXS, TLB Invalidate by ASID, EL1, Inner Shareable

The TLBI ASIDE1IS, TLBI ASIDE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ASIDE1IS, TLBI ASIDE1ISNXS are UNDEFINED.

Attributes

TLBI ASIDE1IS, TLBI ASIDE1ISNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1IS, TLBI ASIDE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ASIDE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIASIDE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr,
                 ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBI_AllAttr,
             ↪X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr, X[t,
             ↪64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr,
                 ↪X[t, 64]);
```

```
When FEAT_XS is implemented TLBI ASIDE1ISNXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIASIDE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
             ↪TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.12 TLBI ASIDE1OS, TLBI ASIDE1OSNXS, TLB Invalidate by ASID, EL1, Outer Shareable

The TLBI ASIDE1OS, TLBI ASIDE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ASIDE1OS, TLBI ASIDE1OSNXS are UNDEFINED.

Attributes

TLBI ASIDE1OS, TLBI ASIDE1OSNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1OS, TLBI ASIDE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ASIDE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIASIDE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr,
                 ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBI_AllAttr,
             ↪X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr, X[t,
             ↪64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                 ↪TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr,
                 ↪X[t, 64]);
```

```
When FEAT_XS is implemented TLBI ASIDE1OSNXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIASIDE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.13 TLBI IPAS2E1, TLBI IPAS2E1NXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI IPAS2E1, TLBI IPAS2E1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2E1, TLBI IPAS2E1NXS are UNDEFINED.

Attributes

TLBI IPAS2E1, TLBI IPAS2E1NXS is a 64-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Executing TLBI IPAS2E1, TLBI IPAS2E1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI IPAS2E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI IPAS2E1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.14 TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS are UNDEFINED.

Attributes

TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI IPAS2E1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.15 TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS are UNDEFINED.

Attributes

TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

Extension to IPA[47:12]. For more information, see IPA[47:12].

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Executing TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI IPAS2E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI IPAS2E1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.16 TLBI IPAS2LE1, TLBI IPAS2LE1NXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI IPAS2LE1, TLBI IPAS2LE1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2LE1, TLBI IPAS2LE1NXS are UNDEFINED.

Attributes

TLBI IPAS2LE1, TLBI IPAS2LE1NXS is a 64-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Executing TLBI IPAS2LE1, TLBI IPAS2LE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI IPAS2LE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI IPAS2LE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.17 TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS are UNDEFINED.

Attributes

TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2LE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI IPAS2LE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.18 TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS are UNDEFINED.

Attributes

TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

Extension to IPA[47:12]. For more information, see IPA[47:12].

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Executing TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI IPAS2LE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI IPAS2LE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.19 TLBI PAALL, TLB Invalidate GPT Information by PA, All Entries, Local

The TLBI PAALL characteristics are:

Purpose

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation applies to all TLB entries containing GPT information.
- The invalidation affects only the TLBs for the PE executing the operation.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI PAALL are UNDEFINED.

Attributes

TLBI PAALL is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI PAALL

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI PAALL{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b100

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_PAALL(Broadcast_NSH);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.20 TLBI PAALLOS, TLB Invalidate GPT Information by PA, All Entries, Outer Shareable

The TLBI PAALLOS characteristics are:

Purpose

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation applies to all TLB entries containing GPT information.
- The invalidation affects all TLBs in the Outer Shareable domain.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI PAALLOS are UNDEFINED.

Attributes

TLBI PAALLOS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI PAALLOS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI PAALLOS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_PAALL(Broadcast_OSH);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.21 TLBI RIPAS2E1, TLBI RIPAS2E1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI RIPAS2E1, TLBI RIPAS2E1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2E1, TLBI RIPAS2E1NXS are UNDEFINED.

Attributes

TLBI RIPAS2E1, TLBI RIPAS2E1NXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RIPAS2E1, TLBI RIPAS2E1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RIPAS2E1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.22 TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RIPAS2E1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.23 TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RIPAS2E1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.24 TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2LE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RIPAS2LE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.25 TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2LE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RIPAS2LE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.26 TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS are UNDEFINED.

Attributes

TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as `0b0000`.

When using a 16KB translation granule, BaseADDR[15:14] is treated as `0b00`.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2LE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI RIPAS2LE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.27 TLBI RPALOS, TLB Range Invalidate GPT Information by PA, Last level, Outer Shareable

The TLBI RPALOS characteristics are:

Purpose

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation affects all TLBs in the Outer Shareable domain.
- Invalidates TLB entries containing GPT information from the final level of the GPT walk that relates to the supplied physical address.
- Invalidations are range-based, invalidating TLB entries starting from the address in BaseADDR, within the range as specified by SIZE.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RPALOS are UNDEFINED.

Attributes

TLBI RPALOS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

SIZE, bits [47:44]

Size of the range for invalidation.

If SIZE is a reserved value, no TLB entries are required to be invalidated.

SIZE	Meaning
0b0000	4KB.
0b0001	16KB.
0b0010	64KB.
0b0011	2MB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIZE	Meaning
0b0100	32MB.
0b0101	512MB.
0b0110	1GB.
0b0111	16GB.
0b1000	64GB.
0b1001	512GB.

All other values are reserved.

If SIZE gives a range smaller than the configured physical granule size in GPCCR_EL3.PGS, then the Effective value of SIZE is taken to be the size configured by GPCCR_EL3.PGS.

If GPCCR_EL3.PGS is configured to a reserved value, no TLB entries are required to be invalidated.

If GPCCR_EL3.PGS is configured to different values at the broadcasting PE and receiving PE, no TLB entries are required to be invalidated at the receiving PE.

Address[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to Address. For more information, see Address.

Otherwise:

Reserved, RES0.

Address, bits [39:0]

The starting address for the range of the maintenance instruction.

This field is decoded with reference to the value of GPCCR_EL3.PGS to give BaseADDR as follows:

GPCCR_EL3.PGS	BaseADDR
0b00 (4KB)	BaseADDR[51:12] = Xt[39:0]
0b10 (16KB)	BaseADDR[51:14] = Xt[39:2]
0b01 (64KB)	BaseADDR[51:16] = Xt[39:4]

Other bits of BaseADDR are treated as zero, to give the Effective value of BaseADDR.

If the Effective value of BaseADDR is not aligned to the size of the Effective value of SIZE, no TLB entries are required to be invalidated.

If the Effective value of BaseADDR targets an address above the implemented PA range that ID_AA64MMFR0_EL1.PARange indicates, no TLB entries are required to be invalidated.

If ID_AA64MMFR0_EL1.PARange is 0b0111, Address[55:52] form the upper part of the BaseADDR value. Otherwise, Address[55:52] are RES0.

Executing TLBI RPALOS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI RPALOS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_RPA(TLBILevel_Last, X[t, 64], Broadcast_OSH);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.28 TLBI RPAOS, TLB Range Invalidate GPT Information by PA, Outer Shareable

The TLBI RPAOS characteristics are:

Purpose

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation affects all TLBs in the Outer Shareable domain.
- Invalidates TLB entries containing GPT information from all levels of the GPT walk that relates to the supplied physical address.
- Invalidations are range-based, invalidating TLB entries starting from the address in BaseADDR, within the range as specified by SIZE.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RPAOS are UNDEFINED.

Attributes

TLBI RPAOS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

SIZE, bits [47:44]

Size of the range for invalidation.

If SIZE is a reserved value, no TLB entries are required to be invalidated.

SIZE	Meaning
0b0000	4KB.
0b0001	16KB.
0b0010	64KB.
0b0011	2MB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIZE	Meaning
0b0100	32MB.
0b0101	512MB.
0b0110	1GB.
0b0111	16GB.
0b1000	64GB.
0b1001	512GB.

All other values are reserved.

If SIZE gives a range smaller than the configured physical granule size in GPCCR_EL3.PGS, then the Effective value of SIZE is taken to be the size configured by GPCCR_EL3.PGS.

If GPCCR_EL3.PGS is configured to a reserved value, no TLB entries are required to be invalidated.

If GPCCR_EL3.PGS is configured to different values at the broadcasting PE and receiving PE, no TLB entries are required to be invalidated at the receiving PE.

Address[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to Address. For more information, see Address.

Otherwise:

Reserved, RES0.

Address, bits [39:0]

The starting address for the range of the maintenance instruction.

This field is decoded with reference to the value of GPCCR_EL3.PGS to give BaseADDR as follows:

GPCCR_EL3.PGS	BaseADDR
0b00 (4KB)	BaseADDR[51:12] = Xt[39:0]
0b10 (16KB)	BaseADDR[51:14] = Xt[39:2]
0b01 (64KB)	BaseADDR[51:16] = Xt[39:4]

Other bits of BaseADDR are treated as zero, to give the Effective value of BaseADDR.

If the Effective value of BaseADDR is not aligned to the size of the Effective value of SIZE, no TLB entries are required to be invalidated.

If the Effective value of BaseADDR targets an address above the implemented PA range that ID_AA64MMFR0_EL1.PARange indicates, no TLB entries are required to be invalidated.

If ID_AA64MMFR0_EL1.PARange is 0b0111, Address[55:52] form the upper part of the BaseADDR value. Otherwise, Address[55:52] are RES0.

Executing TLBI RPAOS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI RPAOS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_RPA(TLBILevel_Any, X[t, 64], Broadcast_OSH);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.29 TLBI RVAAE1, TLBI RVAAE1NXS, TLB Range Invalidate by VA, All ASID, EL1

The TLBI RVAAE1, TLBI RVAAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAAE1, TLBI RVAAE1NXS are UNDEFINED.

Attributes

TLBI RVAAE1, TLBI RVAAE1NXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1, TLBI RVAAE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL0) then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                    ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                    ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAAE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.30 TLBI RVAAE1IS, TLBI RVAAE1ISNXS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI RVAAE1IS, TLBI RVAAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAAE1IS, TLBI RVAAE1ISNXS are UNDEFINED.

Attributes

TLBI RVAAE1IS, TLBI RVAAE1ISNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1IS, TLBI RVAAE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
            ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAAE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.31 TLBI RVAAE1OS, TLBI RVAAE1OSNXS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI RVAAE1OS, TLBI RVAAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAAE1OS, TLBI RVAAE1OSNXS are UNDEFINED.

Attributes

TLBI RVAAE1OS, TLBI RVAAE1OSNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1OS, TLBI RVAAE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAAE1OS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
                ↪HCRX_EL2.FnXS == '1' then
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                    ↪TLBI_ExcludeXS, X[t, 64]);
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                    ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL0) then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                    ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                    ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAAE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.32 TLBI RVAALE1, TLBI RVAALE1NXS, TLB Range Invalidate by VA, All ASID, Last level, EL1

The TLBI RVAALE1, TLBI RVAALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAALE1, TLBI RVAALE1NXS are UNDEFINED.

Attributes

TLBI RVAALE1, TLBI RVAALE1NXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1, TLBI RVAALE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL0) then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                    ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                    ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVAALE1NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.33 TLBI RVAALE1IS, TLBI RVAALE1ISNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI RVAALE1IS, TLBI RVAALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAALE1IS, TLBI RVAALE1ISNXS are UNDEFINED.

Attributes

TLBI RVAALE1IS, TLBI RVAALE1ISNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1IS, TLBI RVAALE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
            ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAALE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.34 TLBI RVAALE1OS, TLBI RVAALE1OSNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBI RVAALE1OS, TLBI RVAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAALE1OS, TLBI RVAALE1OSNXS are UNDEFINED.

Attributes

TLBI RVAALE1OS, TLBI RVAALE1OSNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1OS, TLBI RVAALE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAALE1OS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
                ↪HCRX_EL2.FnXS == '1' then
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                    ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                    ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL0) then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                    ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                    ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVAALE1OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.35 TLBI RVAE1, TLBI RVAE1NXS, TLB Range Invalidate by VA, EL1

The TLBI RVAE1, TLBI RVAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE1, TLBI RVAE1NXS are UNDEFINED.

Attributes

TLBI RVAE1, TLBI RVAE1NXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
`0b00`	Reserved.
`0b01`	4K translation granule.
`0b10`	16K translation granule.
`0b11`	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE1, TLBI RVAE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
            ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVAE1NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.36 TLBI RVAE1IS, TLBI RVAE1ISNXS, TLB Range Invalidate by VA, EL1, Inner Shareable

The TLBI RVAE1IS, TLBI RVAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE1IS, TLBI RVAE1ISNXS are UNDEFINED.

Attributes

TLBI RVAE1IS, TLBI RVAE1ISNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE1IS, TLBI RVAE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
             ↪HFGITR_EL2.TLBIRVAE1IS == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
                 ↪HCRX_EL2.FnXS == '1' then
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                     ↪TLBI_ExcludeXS, X[t, 64]);
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                     ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL0) then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                     ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                     ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIRVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.37 TLBI RVAE1OS, TLBI RVAE1OSNXS, TLB Range Invalidate by VA, EL1, Outer Shareable

The TLBI RVAE1OS, TLBI RVAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE1OS, TLBI RVAE1OSNXS are UNDEFINED.

Attributes

TLBI RVAE1OS, TLBI RVAE1OSNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE1OS, TLBI RVAE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
             ↪HFGITR_EL2.TLBIRVAE1OS == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
                 ↪HCRX_EL2.FnXS == '1' then
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                     ↪TLBI_ExcludeXS, X[t, 64]);
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                     ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL0) then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                     ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                     ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIRVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.38 TLBI RVAE2, TLBI RVAE2NXS, TLB Range Invalidate by VA, EL2

The TLBI RVAE2, TLBI RVAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE2, TLBI RVAE2NXS are UNDEFINED.

Attributes

TLBI RVAE2, TLBI RVAE2NXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2, TLBI RVAE2NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVAE2NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.39 TLBI RVAE2IS, TLBI RVAE2ISNXS, TLB Range Invalidate by VA, EL2, Inner Shareable

The TLBI RVAE2IS, TLBI RVAE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE2IS, TLBI RVAE2ISNXS are UNDEFINED.

Attributes

TLBI RVAE2IS, TLBI RVAE2ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2IS, TLBI RVAE2ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVAE2ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.40 TLBI RVAE2OS, TLBI RVAE2OSNXS, TLB Range Invalidate by VA, EL2, Outer Shareable

The TLBI RVAE2OS, TLBI RVAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE2OS, TLBI RVAE2OSNXS are UNDEFINED.

Attributes

TLBI RVAE2OS, TLBI RVAE2OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2OS, TLBI RVAE2OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAE2OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.41 TLBI RVAE3, TLBI RVAE3NXS, TLB Range Invalidate by VA, EL3

The TLBI RVAE3, TLBI RVAE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE3, TLBI RVAE3NXS are UNDEFINED.

Attributes

TLBI RVAE3, TLBI RVAE3NXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3, TLBI RVAE3NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAE3NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.42 TLBI RVAE3IS, TLBI RVAE3ISNXS, TLB Range Invalidate by VA, EL3, Inner Shareable

The TLBI RVAE3IS, TLBI RVAE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE3IS, TLBI RVAE3ISNXS are UNDEFINED.

Attributes

TLBI RVAE3IS, TLBI RVAE3ISNXS is a 64-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3IS, TLBI RVAE3ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAE3ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.43 TLBI RVAE3OS, TLBI RVAE3OSNXS, TLB Range Invalidate by VA, EL3, Outer Shareable

The TLBI RVAE3OS, TLBI RVAE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE3OS, TLBI RVAE3OSNXS are UNDEFINED.

Attributes

TLBI RVAE3OS, TLBI RVAE3OSNXS is a 64-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3OS, TLBI RVAE3OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVAE3OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH,
        ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.44 TLBI RVALE1, TLBI RVALE1NXS, TLB Range Invalidate by VA, Last level, EL1

The TLBI RVALE1, TLBI RVALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE1, TLBI RVALE1NXS are UNDEFINED.

Attributes

TLBI RVALE1, TLBI RVALE1NXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
`0b00`	Reserved.
`0b01`	4K translation granule.
`0b10`	16K translation granule.
`0b11`	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE1, TLBI RVALE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
            ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVALE1NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.45 TLBI RVALE1IS, TLBI RVALE1ISNXS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI RVALE1IS, TLBI RVALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE1IS, TLBI RVALE1ISNXS are UNDEFINED.

Attributes

TLBI RVALE1IS, TLBI RVALE1ISNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE1IS, TLBI RVALE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
            ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVALE1ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
        ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if ELIsInHost(EL0) then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
else
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.46 TLBI RVALE1OS, TLBI RVALE1OSNXS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI RVALE1OS, TLBI RVALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE1OS, TLBI RVALE1OSNXS are UNDEFINED.

Attributes

TLBI RVALE1OS, TLBI RVALE1OSNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE1OS, TLBI RVALE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVALE1OS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
                ↪HCRX_EL2.FnXS == '1' then
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                    ↪TLBI_ExcludeXS, X[t, 64]);
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                    ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL0) then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                    ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                    ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVALE1OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b101

```
    if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
        ↪IsFeatureImplemented(FEAT_AA64)) then
        UNDEFINED;
    elsif !IsFeatureImplemented(FEAT_XS) then
        UNDEFINED;
    elsif PSTATE.EL == EL0 then
        UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.TTLB == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
            ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
            ↪HFGITR_EL2.TLBIRVALE1OS == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL0) then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.47 TLBI RVALE2, TLBI RVALE2NXS, TLB Range Invalidate by VA, Last level, EL2

The TLBI RVALE2, TLBI RVALE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE2, TLBI RVALE2NXS are UNDEFINED.

Attributes

TLBI RVALE2, TLBI RVALE2NXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2, TLBI RVALE2NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVALE2NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.48 TLBI RVALE2IS, TLBI RVALE2ISNXS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI RVALE2IS, TLBI RVALE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE2IS, TLBI RVALE2ISNXS are UNDEFINED.

Attributes

TLBI RVALE2IS, TLBI RVALE2ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2IS, TLBI RVALE2ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVALE2ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.49 TLBI RVALE2OS, TLBI RVALE2OSNXS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI RVALE2OS, TLBI RVALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE2OS, TLBI RVALE2OSNXS are UNDEFINED.

Attributes

TLBI RVALE2OS, TLBI RVALE2OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2OS, TLBI RVALE2OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI RVALE2OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.50 TLBI RVALE3, TLBI RVALE3NXS, TLB Range Invalidate by VA, Last level, EL3

The TLBI RVALE3, TLBI RVALE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE3, TLBI RVALE3NXS are UNDEFINED.

Attributes

TLBI RVALE3, TLBI RVALE3NXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3, TLBI RVALE3NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVALE3NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.51 TLBI RVALE3IS, TLBI RVALE3ISNXS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI RVALE3IS, TLBI RVALE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE3IS, TLBI RVALE3ISNXS are UNDEFINED.

Attributes

TLBI RVALE3IS, TLBI RVALE3ISNXS is a 64-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3IS, TLBI RVALE3ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVALE3ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.52 TLBI RVALE3OS, TLBI RVALE3OSNXS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable

The TLBI RVALE3OS, TLBI RVALE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE3OS, TLBI RVALE3OSNXS are UNDEFINED.

Attributes

TLBI RVALE3OS, TLBI RVALE3OSNXS is a 64-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3OS, TLBI RVALE3OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI RVALE3OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH,
        ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.53 TLBI VAAE1, TLBI VAAE1NXS, TLB Invalidate by VA, All ASID, EL1

The TLBI VAAE1, TLBI VAAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAAE1, TLBI VAAE1NXS are UNDEFINED.

Attributes

TLBI VAAE1, TLBI VAAE1NXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1, TLBI VAAE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI VAAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VAAE1NXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.54 TLBI VAAE1IS, TLBI VAAE1ISNXS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAAE1IS, TLBI VAAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAAE1IS, TLBI VAAE1ISNXS are UNDEFINED.

Attributes

TLBI VAAE1IS, TLBI VAAE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1IS, TLBI VAAE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI VAAE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.55 TLBI VAAE1OS, TLBI VAAE1OSNXS, TLB Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI VAAE1OS, TLBI VAAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAAE1OS, TLBI VAAE1OSNXS are UNDEFINED.

Attributes

TLBI VAAE1OS, TLBI VAAE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1OS, TLBI VAAE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI VAAE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.56 TLBI VAALE1, TLBI VAALE1NXS, TLB Invalidate by VA, All ASID, Last level, EL1

The TLBI VAALE1, TLBI VAALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAALE1, TLBI VAALE1NXS are UNDEFINED.

Attributes

TLBI VAALE1, TLBI VAALE1NXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1, TLBI VAALE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI VAALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VAALE1NXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.57 TLBI VAALE1IS, TLBI VAALE1ISNXS, TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI VAALE1IS, TLBI VAALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAALE1IS, TLBI VAALE1ISNXS are UNDEFINED.

Attributes

TLBI VAALE1IS, TLBI VAALE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1IS, TLBI VAALE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI VAALE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.58 TLBI VAALE1OS, TLBI VAALE1OSNXS, TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBI VAALE1OS, TLBI VAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAALE1OS, TLBI VAALE1OSNXS are UNDEFINED.

Attributes

TLBI VAALE1OS, TLBI VAALE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1OS, TLBI VAALE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI VAALE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.59 TLBI VAE1, TLBI VAE1NXS, TLB Invalidate by VA, EL1

The TLBI VAE1, TLBI VAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE1, TLBI VAE1NXS are UNDEFINED.

Attributes

TLBI VAE1, TLBI VAE1NXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VAE1, TLBI VAE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VAE1NXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.60 TLBI VAE1IS, TLBI VAE1ISNXS, TLB Invalidate by VA, EL1, Inner Shareable

The TLBI VAE1IS, TLBI VAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE1IS, TLBI VAE1ISNXS are UNDEFINED.

Attributes

TLBI VAE1IS, TLBI VAE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE1IS, TLBI VAE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VAE1ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.61 TLBI VAE1OS, TLBI VAE1OSNXS, TLB Invalidate by VA, EL1, Outer Shareable

The TLBI VAE1OS, TLBI VAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE1OS, TLBI VAE1OSNXS are UNDEFINED.

Attributes

TLBI VAE1OS, TLBI VAE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE1OS, TLBI VAE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VAE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.62 TLBI VAE2, TLBI VAE2NXS, TLB Invalidate by VA, EL2

The TLBI VAE2, TLBI VAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE2, TLBI VAE2NXS are UNDEFINED.

Attributes

TLBI VAE2, TLBI VAE2NXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VAE2, TLBI VAE2NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VAE2NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.63 TLBI VAE2IS, TLBI VAE2ISNXS, TLB Invalidate by VA, EL2, Inner Shareable

The TLBI VAE2IS, TLBI VAE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE2IS, TLBI VAE2ISNXS are UNDEFINED.

Attributes

TLBI VAE2IS, TLBI VAE2ISNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VAE2IS, TLBI VAE2ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VAE2ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.64 TLBI VAE2OS, TLBI VAE2OSNXS, TLB Invalidate by VA, EL2, Outer Shareable

The TLBI VAE2OS, TLBI VAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE2OS, TLBI VAE2OSNXS are UNDEFINED.

Attributes

TLBI VAE2OS, TLBI VAE2OSNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE2OS, TLBI VAE2OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VAE2OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.65 TLBI VAE3, TLBI VAE3NXS, TLB Invalidate by VA, EL3

The TLBI VAE3, TLBI VAE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE3, TLBI VAE3NXS are UNDEFINED.

Attributes

TLBI VAE3, TLBI VAE3NXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3, TLBI VAE3NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0111`	`0b001`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VAE3NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.66 TLBI VAE3IS, TLBI VAE3ISNXS, TLB Invalidate by VA, EL3, Inner Shareable

The TLBI VAE3IS, TLBI VAE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE3IS, TLBI VAE3ISNXS are UNDEFINED.

Attributes

TLBI VAE3IS, TLBI VAE3ISNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3IS, TLBI VAE3ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0011`	`0b001`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VAE3ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.67 TLBI VAE3OS, TLBI VAE3OSNXS, TLB Invalidate by VA, EL3, Outer Shareable

The TLBI VAE3OS, TLBI VAE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE3OS, TLBI VAE3OSNXS are UNDEFINED.

Attributes

TLBI VAE3OS, TLBI VAE3OSNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3OS, TLBI VAE3OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0001`	`0b001`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VAE3OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.68 TLBI VALE1, TLBI VALE1NXS, TLB Invalidate by VA, Last level, EL1

The TLBI VALE1, TLBI VALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE1, TLBI VALE1NXS are UNDEFINED.

Attributes

TLBI VALE1, TLBI VALE1NXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VALE1, TLBI VALE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VALE1NXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.69 TLBI VALE1IS, TLBI VALE1ISNXS, TLB Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI VALE1IS, TLBI VALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE1IS, TLBI VALE1ISNXS are UNDEFINED.

Attributes

TLBI VALE1IS, TLBI VALE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE1IS, TLBI VALE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VALE1ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.70 TLBI VALE1OS, TLBI VALE1OSNXS, TLB Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI VALE1OS, TLBI VALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE1OS, TLBI VALE1OSNXS are UNDEFINED.

Attributes

TLBI VALE1OS, TLBI VALE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE1OS, TLBI VALE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VALE1OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.71 TLBI VALE2, TLBI VALE2NXS, TLB Invalidate by VA, Last level, EL2

The TLBI VALE2, TLBI VALE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE2, TLBI VALE2NXS are UNDEFINED.

Attributes

TLBI VALE2, TLBI VALE2NXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2, TLBI VALE2NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VALE2NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.72 TLBI VALE2IS, TLBI VALE2ISNXS, TLB Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI VALE2IS, TLBI VALE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE2IS, TLBI VALE2ISNXS are UNDEFINED.

Attributes

TLBI VALE2IS, TLBI VALE2ISNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2IS, TLBI VALE2ISNXS

This system instruction is an alias of the SYS instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VALE2ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.73 TLBI VALE2OS, TLBI VALE2OSNXS, TLB Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI VALE2OS, TLBI VALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE2OS, TLBI VALE2OSNXS are UNDEFINED.

Attributes

TLBI VALE2OS, TLBI VALE2OSNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2OS, TLBI VALE2OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VALE2OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.74 TLBI VALE3, TLBI VALE3NXS, TLB Invalidate by VA, Last level, EL3

The TLBI VALE3, TLBI VALE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE3, TLBI VALE3NXS are UNDEFINED.

Attributes

TLBI VALE3, TLBI VALE3NXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3, TLBI VALE3NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0111`	`0b101`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VALE3NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.75 TLBI VALE3IS, TLBI VALE3ISNXS, TLB Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI VALE3IS, TLBI VALE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE3IS, TLBI VALE3ISNXS are UNDEFINED.

Attributes

TLBI VALE3IS, TLBI VALE3ISNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3IS, TLBI VALE3ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0011`	`0b101`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VALE3ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.76 TLBI VALE3OS, TLBI VALE3OSNXS, TLB Invalidate by VA, Last level, EL3, Outer Shareable

The TLBI VALE3OS, TLBI VALE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE3OS, TLBI VALE3OSNXS are UNDEFINED.

Attributes

TLBI VALE3OS, TLBI VALE3OSNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3OS, TLBI VALE3OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0001`	`0b101`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
```

When FEAT_XS is implemented TLBI VALE3OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.77 TLBI VMALLE1, TLBI VMALLE1NXS, TLB Invalidate by VMID, All at stage 1, EL1

The TLBI VMALLE1, TLBI VMALLE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLE1, TLBI VMALLE1NXS are UNDEFINED.

Attributes

TLBI VMALLE1, TLBI VMALLE1NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLE1, TLBI VMALLE1NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI VMALLE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVMALLE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr,
                 ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBI_AllAttr,
             ↪X[t, 64]);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr,
             ↪X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr,
                 ↪X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VMALLE1NXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVMALLE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
             ↪TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.78 TLBI VMALLE1IS, TLBI VMALLE1ISNXS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable

The TLBI VMALLE1IS, TLBI VMALLE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLE1IS, TLBI VMALLE1ISNXS are UNDEFINED.

Attributes

TLBI VMALLE1IS, TLBI VMALLE1ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VMALLE1IS, TLBI VMALLE1ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVMALLE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr,
                 ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBI_AllAttr,
             ↪X[t, 64]);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr,
             ↪X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr,
                 ↪X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VMALLE1ISNXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVMALLE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
             ↪TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.79 TLBI VMALLE1OS, TLBI VMALLE1OSNXS, TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable

The TLBI VMALLE1OS, TLBI VMALLE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLE1OS, TLBI VMALLE1OSNXS are UNDEFINED.

Attributes

TLBI VMALLE1OS, TLBI VMALLE1OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VMALLE1OS, TLBI VMALLE1OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVMALLE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr,
                 ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBI_AllAttr,
             ↪X[t, 64]);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr,
             ↪X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                 ↪TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr,
                 ↪X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VMALLE1OSNXS{, <Xt>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
         ↪&& IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVMALLE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
             ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.80 TLBI VMALLS12E1, TLBI VMALLS12E1NXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1

The TLBI VMALLS12E1, TLBI VMALLS12E1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLS12E1, TLBI VMALLS12E1NXS are UNDEFINED.

Attributes

TLBI VMALLS12E1, TLBI VMALLS12E1NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1, TLBI VMALLS12E1NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VMALLS12E1{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr, X[t,
        ↪64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_NSH, TLBI_AllAttr,
            ↪X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBI_AllAttr, X[t, 64]);
```

`When FEAT_XS is implemented TLBI VMALLS12E1NXS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_NSH,
            ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.81 TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable

The TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS are UNDEFINED.

Attributes

TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLS12E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr, X[t,
        ↪64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_ISH, TLBI_AllAttr,
            ↪X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VMALLS12E1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_ISH,
            ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.82 TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable

The TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS are UNDEFINED.

Attributes

TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS is a 64-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLS12E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr, X[t,
        ↪64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_OSH, TLBI_AllAttr,
            ↪X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VMALLS12E1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
        ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_OSH,
            ↪TLBI_ExcludeXS, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.83 TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS, TLB Invalidate stage 2 dirty state by VMID, EL1&0

The TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS characteristics are:

Purpose

Invalidates stage 2 write permissions from cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry would be used for stage 2 translation. This applies if the TLB entry holds information from stage 2 translation only, or combined information from stage 1 and stage 2 translation.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIW is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS are UNDEFINED.

Attributes

TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS is a 64-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:0]

Reserved, RES0.

Executing TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLWS2E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr, X[t,
        ↪64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VMALLWS2E1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
         ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.84 TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS, TLB Invalidate stage 2 dirty state by VMID, EL1&0, Inner Shareable

The TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS characteristics are:

Purpose

Invalidates stage 2 write permissions from cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry would be used for stage 2 translation. This applies if the TLB entry holds information from stage 2 translation only, or combined information from stage 1 and stage 2 translation.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIW is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS are UNDEFINED.

Attributes

TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLWS2E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr, X[t,
        ↪64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VMALLWS2E1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
         ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.85 TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS, TLB Invalidate stage 2 write permission by VMID, EL1&0, Outer Shareable

The TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS characteristics are:

Purpose

Invalidates stage 2 write permissions from cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry would be used for stage 2 translation. This applies if the TLB entry holds information from stage 2 translation only, or combined information from stage 1 and stage 2 translation.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIW is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS are UNDEFINED.

Attributes

TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLWS2E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr, X[t,
        ↪64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBI_AllAttr, X[t, 64]);
```

```
When FEAT_XS is implemented TLBI VMALLWS2E1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
         ↪TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBI_ExcludeXS, X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.86 TLBIP IPAS2E1, TLBIP IPAS2E1NXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1

The TLBIP IPAS2E1, TLBIP IPAS2E1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2E1, TLBIP IPAS2E1NXS are UNDEFINED.

Attributes

TLBIP IPAS2E1, TLBIP IPAS2E1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2E1, TLBIP IPAS2E1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2E1{, <Xt>, <Xt2>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP IPAS2E1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.87 TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS are UNDEFINED.

Attributes

TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2E1IS{, <Xt>, <Xt2>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

TLBIP IPAS2E1ISNXS{, <Xt>, <Xt2>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.88 TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS are UNDEFINED.

Attributes

TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2E1OS{, <Xt>, <Xt2>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP IPAS2E1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.89 TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS are UNDEFINED.

Attributes

TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2LE1{, <Xt>, <Xt2>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP IPAS2LE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.90 TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS are UNDEFINED.

Attributes

TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2LE1IS{, <Xt>, <Xt2>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP IPAS2LE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.91 TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS are UNDEFINED.

Attributes

TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2LE1OS{, <Xt>, <Xt2>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP IPAS2LE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.92 TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS are UNDEFINED.

Attributes

TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2E1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RIPAS2E1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.93 TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS are UNDEFINED.

Attributes

TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2E1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RIPAS2E1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.94 TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS are UNDEFINED.

Attributes

TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2E1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RIPAS2E1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.95 TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS are UNDEFINED.

Attributes

TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2LE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b110

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RIPAS2LE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b110

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.96 TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS are UNDEFINED.

Attributes

TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2LE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RIPAS2LE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.97 TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS are UNDEFINED.

Attributes

TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RIPAS2LE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RIPAS2LE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.98 TLBIP RVAAE1, TLBIP RVAAE1NXS, TLB Range Invalidate by VA, All ASID, EL1

The TLBIP RVAAE1, TLBIP RVAAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAAE1, TLBIP RVAAE1NXS are UNDEFINED.

Attributes

TLBIP RVAAE1, TLBIP RVAAE1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAAE1, TLBIP RVAAE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAAE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                    ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                    ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAAE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
    ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.99 TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAAE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAAE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIRVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.100 TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAAE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAAE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIRVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.101 TLBIP RVAALE1, TLBIP RVAALE1NXS, TLB Range Invalidate by VA, All ASID, Last level, EL1

The TLBIP RVAALE1, TLBIP RVAALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAALE1, TLBIP RVAALE1NXS are UNDEFINED.

Attributes

TLBIP RVAALE1, TLBIP RVAALE1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAALE1, TLBIP RVAALE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAALE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, t2, 128]);
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL0) then
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
                return;
            else
                AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                    ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
        else
            if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
                return;
            else
                AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                    ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAALE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
    ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.102 TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAALE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
             ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAALE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIRVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.103 TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAALE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
             ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAALE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIRVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.104 TLBIP RVAE1, TLBIP RVAE1NXS, TLB Range Invalidate by VA, EL1

The TLBIP RVAE1, TLBIP RVAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE1, TLBIP RVAE1NXS are UNDEFINED.

Attributes

TLBIP RVAE1, TLBIP RVAE1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE1, TLBIP RVAE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.105 TLBIP RVAE1IS, TLBIP RVAE1ISNXS, TLB Range Invalidate by VA, EL1, Inner Shareable

The TLBIP RVAE1IS, TLBIP RVAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE1IS, TLBIP RVAE1ISNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAE1IS, TLBIP RVAE1ISNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE1IS, TLBIP RVAE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.106 TLBIP RVAE1OS, TLBIP RVAE1OSNXS, TLB Range Invalidate by VA, EL1, Outer Shareable

The TLBIP RVAE1OS, TLBIP RVAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE1OS, TLBIP RVAE1OSNXS are UNDEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAE1OS, TLBIP RVAE1OSNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE1OS, TLBIP RVAE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.107 TLBIP RVAE2, TLBIP RVAE2NXS, TLB Range Invalidate by VA, EL2

The TLBIP RVAE2, TLBIP RVAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE2, TLBIP RVAE2NXS are UNDEFINED.

Attributes

TLBIP RVAE2, TLBIP RVAE2NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE2, TLBIP RVAE2NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE2{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE2NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.108 TLBIP RVAE2IS, TLBIP RVAE2ISNXS, TLB Range Invalidate by VA, EL2, Inner Shareable

The TLBIP RVAE2IS, TLBIP RVAE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE2IS, TLBIP RVAE2ISNXS are UNDEFINED.

Attributes

TLBIP RVAE2IS, TLBIP RVAE2ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE2IS, TLBIP RVAE2ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE2IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE2ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.109 TLBIP RVAE2OS, TLBIP RVAE2OSNXS, TLB Range Invalidate by VA, EL2, Outer Shareable

The TLBIP RVAE2OS, TLBIP RVAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE2OS, TLBIP RVAE2OSNXS are UNDEFINED.

Attributes

TLBIP RVAE2OS, TLBIP RVAE2OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE2OS, TLBIP RVAE2OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE2OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE2OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.110 TLBIP RVAE3, TLBIP RVAE3NXS, TLB Range Invalidate by VA, EL3

The TLBIP RVAE3, TLBIP RVAE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE3, TLBIP RVAE3NXS are UNDEFINED.

Attributes

TLBIP RVAE3, TLBIP RVAE3NXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE3, TLBIP RVAE3NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVAE3{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE3NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.111 TLBIP RVAE3IS, TLBIP RVAE3ISNXS, TLB Range Invalidate by VA, EL3, Inner Shareable

The TLBIP RVAE3IS, TLBIP RVAE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE3IS, TLBIP RVAE3ISNXS are UNDEFINED.

Attributes

TLBIP RVAE3IS, TLBIP RVAE3ISNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE3IS, TLBIP RVAE3ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVAE3IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE3ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.112 TLBIP RVAE3OS, TLBIP RVAE3OSNXS, TLB Range Invalidate by VA, EL3, Outer Shareable

The TLBIP RVAE3OS, TLBIP RVAE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE3OS, TLBIP RVAE3OSNXS are UNDEFINED.

Attributes

TLBIP RVAE3OS, TLBIP RVAE3OSNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE3OS, TLBIP RVAE3OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVAE3OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVAE3OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.113 TLBIP RVALE1, TLBIP RVALE1NXS, TLB Range Invalidate by VA, Last level, EL1

The TLBIP RVALE1, TLBIP RVALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE1, TLBIP RVALE1NXS are UNDEFINED.

Attributes

TLBIP RVALE1, TLBIP RVALE1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE1, TLBIP RVALE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIRVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.114 TLBIP RVALE1IS, TLBIP RVALE1ISNXS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable

The TLBIP RVALE1IS, TLBIP RVALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE1IS, TLBIP RVALE1ISNXS are UNDEFINED.

Attributes

TLBIP RVALE1IS, TLBIP RVALE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE1IS, TLBIP RVALE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIRVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.115 TLBIP RVALE1OS, TLBIP RVALE1OSNXS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable

The TLBIP RVALE1OS, TLBIP RVALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE1OS, TLBIP RVALE1OSNXS are UNDEFINED.

Attributes

TLBIP RVALE1OS, TLBIP RVALE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE1OS, TLBIP RVALE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIRVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIRVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.116 TLBIP RVALE2, TLBIP RVALE2NXS, TLB Range Invalidate by VA, Last level, EL2

The TLBIP RVALE2, TLBIP RVALE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE2, TLBIP RVALE2NXS are UNDEFINED.

Attributes

TLBIP RVALE2, TLBIP RVALE2NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE2, TLBIP RVALE2NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE2{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE2NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.117 TLBIP RVALE2IS, TLBIP RVALE2ISNXS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable

The TLBIP RVALE2IS, TLBIP RVALE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE2IS, TLBIP RVALE2ISNXS are UNDEFINED.

Attributes

TLBIP RVALE2IS, TLBIP RVALE2ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE2IS, TLBIP RVALE2ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE2IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE2ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.118 TLBIP RVALE2OS, TLBIP RVALE2OSNXS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable

The TLBIP RVALE2OS, TLBIP RVALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE2OS, TLBIP RVALE2OSNXS are UNDEFINED.

Attributes

TLBIP RVALE2OS, TLBIP RVALE2OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE2OS, TLBIP RVALE2OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE2OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE2OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.119 TLBIP RVALE3, TLBIP RVALE3NXS, TLB Range Invalidate by VA, Last level, EL3

The TLBIP RVALE3, TLBIP RVALE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE3, TLBIP RVALE3NXS are UNDEFINED.

Attributes

TLBIP RVALE3, TLBIP RVALE3NXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE3, TLBIP RVALE3NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVALE3{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE3NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.120 TLBIP RVALE3IS, TLBIP RVALE3ISNXS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable

The TLBIP RVALE3IS, TLBIP RVALE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE3IS, TLBIP RVALE3ISNXS are UNDEFINED.

Attributes

TLBIP RVALE3IS, TLBIP RVALE3ISNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE3IS, TLBIP RVALE3ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVALE3IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE3ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.121 TLBIP RVALE3OS, TLBIP RVALE3OSNXS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable

The TLBIP RVALE3OS, TLBIP RVALE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE3OS, TLBIP RVALE3OSNXS are UNDEFINED.

Attributes

TLBIP RVALE3OS, TLBIP RVALE3OSNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE3OS, TLBIP RVALE3OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVALE3OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP RVALE3OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.122 TLBIP VAAE1, TLBIP VAAE1NXS, TLB Invalidate Pair by VA, All ASID, EL1

The TLBIP VAAE1, TLBIP VAAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAAE1, TLBIP VAAE1NXS are UNDEFINED.

Attributes

TLBIP VAAE1, TLBIP VAAE1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAAE1, TLBIP VAAE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAAE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0111`	`0b011`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAAE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.123 TLBIP VAAE1IS, TLBIP VAAE1ISNXS, TLB Invalidate Pair by VA, All ASID, EL1, Inner Shareable

The TLBIP VAAE1IS, TLBIP VAAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAAE1IS, TLBIP VAAE1ISNXS are UNDEFINED.

Attributes

TLBIP VAAE1IS, TLBIP VAAE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAAE1IS, TLBIP VAAE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAAE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0011`	`0b011`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAAE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
    ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.124 TLBIP VAAE1OS, TLBIP VAAE1OSNXS, TLB Invalidate Pair by VA, All ASID, EL1, Outer Shareable

The TLBIP VAAE1OS, TLBIP VAAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAAE1OS, TLBIP VAAE1OSNXS are UNDEFINED.

Attributes

TLBIP VAAE1OS, TLBIP VAAE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAAE1OS, TLBIP VAAE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAAE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAAE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
    ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.125 TLBIP VAALE1, TLBIP VAALE1NXS, TLB Invalidate Pair by VA, All ASID, Last level, EL1

The TLBIP VAALE1, TLBIP VAALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAALE1, TLBIP VAALE1NXS are UNDEFINED.

Attributes

TLBIP VAALE1, TLBIP VAALE1NXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAALE1, TLBIP VAALE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAALE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0111`	`0b111`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAALE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.126 TLBIP VAALE1IS, TLBIP VAALE1ISNXS, TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBIP VAALE1IS, TLBIP VAALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAALE1IS, TLBIP VAALE1ISNXS are UNDEFINED.

Attributes

TLBIP VAALE1IS, TLBIP VAALE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAALE1IS, TLBIP VAALE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAALE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0011`	`0b111`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAALE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH,
    ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.127 TLBIP VAALE1OS, TLBIP VAALE1OSNXS, TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBIP VAALE1OS, TLBIP VAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAALE1OS, TLBIP VAALE1OSNXS are UNDEFINED.

Attributes

TLBIP VAALE1OS, TLBIP VAALE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAALE1OS, TLBIP VAALE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAALE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAALE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH,
    ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.128 TLBIP VAE1, TLBIP VAE1NXS, TLB Invalidate Pair by VA, EL1

The TLBIP VAE1, TLBIP VAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE1, TLBIP VAE1NXS are UNDEFINED.

Attributes

TLBIP VAE1, TLBIP VAE1NXS is a 128-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE1, TLBIP VAE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                 ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
             ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.129 TLBIP VAE1IS, TLBIP VAE1ISNXS, TLB Invalidate Pair by VA, EL1, Inner Shareable

The TLBIP VAE1IS, TLBIP VAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE1IS, TLBIP VAE1ISNXS are UNDEFINED.

Attributes

TLBIP VAE1IS, TLBIP VAE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE1IS, TLBIP VAE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0011`	`0b001`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.130 TLBIP VAE1OS, TLBIP VAE1OSNXS, TLB Invalidate Pair by VA, EL1, Outer Shareable

The TLBIP VAE1OS, TLBIP VAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE1OS, TLBIP VAE1OSNXS are UNDEFINED.

Attributes

TLBIP VAE1OS, TLBIP VAE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE1OS, TLBIP VAE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0001`	`0b001`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.131 TLBIP VAE2, TLBIP VAE2NXS, TLB Invalidate Pair by VA, EL2

The TLBIP VAE2, TLBIP VAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE2, TLBIP VAE2NXS are UNDEFINED.

Attributes

TLBIP VAE2, TLBIP VAE2NXS is a 128-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE2, TLBIP VAE2NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE2{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE2NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.132 TLBIP VAE2IS, TLBIP VAE2ISNXS, TLB Invalidate Pair by VA, EL2, Inner Shareable

The TLBIP VAE2IS, TLBIP VAE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE2IS, TLBIP VAE2ISNXS are UNDEFINED.

Attributes

TLBIP VAE2IS, TLBIP VAE2ISNXS is a 128-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE2IS, TLBIP VAE2ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE2IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE2ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b001

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.133 TLBIP VAE2OS, TLBIP VAE2OSNXS, TLB Invalidate Pair by VA, EL2, Outer Shareable

The TLBIP VAE2OS, TLBIP VAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE2OS, TLBIP VAE2OSNXS are UNDEFINED.

Attributes

TLBIP VAE2OS, TLBIP VAE2OSNXS is a 128-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE2OS, TLBIP VAE2OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE2OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b100`	`0b1000`	`0b0001`	`0b001`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE2OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.134 TLBIP VAE3, TLBIP VAE3NXS, TLB Invalidate Pair by VA, EL3

The TLBIP VAE3, TLBIP VAE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE3, TLBIP VAE3NXS are UNDEFINED.

Attributes

TLBIP VAE3, TLBIP VAE3NXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE3, TLBIP VAE3NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VAE3{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE3NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.135 TLBIP VAE3IS, TLBIP VAE3ISNXS, TLB Invalidate Pair by VA, EL3, Inner Shareable

The TLBIP VAE3IS, TLBIP VAE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE3IS, TLBIP VAE3ISNXS are UNDEFINED.

Attributes

TLBIP VAE3IS, TLBIP VAE3ISNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE3IS, TLBIP VAE3ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VAE3IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE3ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.136 TLBIP VAE3OS, TLBIP VAE3OSNXS, TLB Invalidate Pair by VA, EL3, Outer Shareable

The TLBIP VAE3OS, TLBIP VAE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE3OS, TLBIP VAE3OSNXS are UNDEFINED.

Attributes

TLBIP VAE3OS, TLBIP VAE3OSNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE3OS, TLBIP VAE3OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VAE3OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VAE3OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.137 TLBIP VALE1, TLBIP VALE1NXS, TLB Invalidate Pair by VA, Last level, EL1

The TLBIP VALE1, TLBIP VALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE1, TLBIP VALE1NXS are UNDEFINED.

Attributes

TLBIP VALE1, TLBIP VALE1NXS is a 128-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE1, TLBIP VALE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                 ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                 ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
         ↪HFGITR_EL2.TLBIVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.138 TLBIP VALE1IS, TLBIP VALE1ISNXS, TLB Invalidate Pair by VA, Last level, EL1, Inner Shareable

The TLBIP VALE1IS, TLBIP VALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE1IS, TLBIP VALE1ISNXS are UNDEFINED.

Attributes

TLBIP VALE1IS, TLBIP VALE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE1IS, TLBIP VALE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0011`	`0b101`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.139 TLBIP VALE1OS, TLBIP VALE1OSNXS, TLB Invalidate Pair by VA, Last level, EL1, Outer Shareable

The TLBIP VALE1OS, TLBIP VALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE1OS, TLBIP VALE1OSNXS are UNDEFINED.

Attributes

TLBIP VALE1OS, TLBIP VALE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE1OS, TLBIP VALE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0001`	`0b101`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.TLBIVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
             ↪HCRX_EL2.FnXS == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') &&
        ↪HFGITR_EL2.TLBIVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL0) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
                ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.140 TLBIP VALE2, TLBIP VALE2NXS, TLB Invalidate Pair by VA, Last level, EL2

The TLBIP VALE2, TLBIP VALE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE2, TLBIP VALE2NXS are UNDEFINED.

Attributes

TLBIP VALE2, TLBIP VALE2NXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE2, TLBIP VALE2NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE2{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE2NXS{, <Xt>, <Xt2>}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.141 TLBIP VALE2IS, TLBIP VALE2ISNXS, TLB Invalidate Pair by VA, Last level, EL2, Inner Shareable

The TLBIP VALE2IS, TLBIP VALE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE2IS, TLBIP VALE2ISNXS are UNDEFINED.

Attributes

TLBIP VALE2IS, TLBIP VALE2ISNXS is a 128-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE2IS, TLBIP VALE2ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE2IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE2ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b101

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.142 TLBIP VALE2OS, TLBIP VALE2OSNXS, TLB Invalidate Pair by VA, Last level, EL2, Outer Shareable

The TLBIP VALE2OS, TLBIP VALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE2OS, TLBIP VALE2OSNXS are UNDEFINED.

Attributes

TLBIP VALE2OS, TLBIP VALE2OSNXS is a 128-bit System instruction.

Field descriptions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE2OS, TLBIP VALE2OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE2OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b100`	`0b1000`	`0b0001`	`0b101`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE2OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64.TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.143 TLBIP VALE3, TLBIP VALE3NXS, TLB Invalidate Pair by VA, Last level, EL3

The TLBIP VALE3, TLBIP VALE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE3, TLBIP VALE3NXS are UNDEFINED.

Attributes

TLBIP VALE3, TLBIP VALE3NXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE3, TLBIP VALE3NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VALE3{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE3NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.144 TLBIP VALE3IS, TLBIP VALE3ISNXS, TLB Invalidate Pair by VA, Last level, EL3, Inner Shareable

The TLBIP VALE3IS, TLBIP VALE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE3IS, TLBIP VALE3ISNXS are UNDEFINED.

Attributes

TLBIP VALE3IS, TLBIP VALE3ISNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE3IS, TLBIP VALE3ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VALE3IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE3ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.145 TLBIP VALE3OS, TLBIP VALE3OSNXS, TLB Invalidate Pair by VA, Last level, EL3, Outer Shareable

The TLBIP VALE3OS, TLBIP VALE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE3OS, TLBIP VALE3OSNXS are UNDEFINED.

Attributes

TLBIP VALE3OS, TLBIP VALE3OSNXS is a 128-bit System instruction.

Field descriptions

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE3OS, TLBIP VALE3OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VALE3OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, t2, 128]);
```

```
TLBIP VALE3OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64.TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X[t, t2, 128]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6 A64 System instructions for prediction restriction

This section lists the A64 System instructions for prediction restriction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.1 CFP RCTX, Control Flow Prediction Restriction by Context

The CFP RCTX characteristics are:

Purpose

Control Flow Prediction Restriction by Context applies to all Control Flow Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_SPECRES is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CFP RCTX are UNDEFINED.

Attributes

CFP RCTX is a 64-bit System instruction.

Field descriptions

Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see CFP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This instruction is treated as a NOP when executed at EL3 and either:

- CFP_RCTX.{NSE, NS} selects a reserved value.
- CFP_RCTX.{NSE, NS} == {1, 0} and CFP_RCTX.EL has a value other than `0b11`.

Otherwise:

Security State.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
`0b00`	EL0.	
`0b01`	EL1	
`0b10`	EL2	FEAT_EL2 is implemented
`0b11`	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
`0b0`	Applies to specified ASID for an EL0 target execution context.
`0b1`	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing CFP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
CFP RCTX, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b100

```
if !(IsFeatureImplemented(FEAT_SPECRES) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.EnRCTX == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.CFPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.EnRCTX == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.CFPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
elsif PSTATE.EL == EL2 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
elsif PSTATE.EL == EL3 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.2 COSP RCTX, Clear Other Speculative Prediction Restriction by Context

The COSP RCTX characteristics are:

Purpose

Clear Other Speculative Prediction Restriction by Context applies to all prediction resources not managed by other speculation restriction System instructions.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control any predictions occurring after the instruction is complete and synchronized.

This instruction applies to all speculative access except:

- Cache Prefetch predictions.
- Control Flow predictions.
- Data Value predictions.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the PE that executed the original restriction instruction, and a subsequent Context Synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations, the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.
This instruction applies to speculative accesses resulting from address predictions.

Configuration

This system instruction is present only when FEAT_SPECRES2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to COSP RCTX are UNDEFINED.

Attributes

COSP RCTX is a 64-bit System instruction.

Field descriptions

Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 and EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see COSP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

This instruction is treated as a NOP when executed at EL3 and either:

- COSP_RCTX.{NSE, NS} selects a reserved value.
- COSP_RCTX.{NSE, NS} == {1, 0} and COSP_RCTX.EL has a value other than `0b11`.

Otherwise:

Security State.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
`0b00`	EL0.	
`0b01`	EL1	
`0b10`	EL2	FEAT_EL2 is implemented
`0b11`	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs, or a specified ASID.

GASID	Meaning
`0b0`	Applies to specified ASID for an EL0 target execution context.
`0b1`	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [15:0]

Only applies to an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing COSP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
COSP RCTX, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b110

```
if !(IsFeatureImplemented(FEAT_SPECRES2) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.EnRCTX == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.COSPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.EnRCTX == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_Other);
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.COSPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_Other);
elsif PSTATE.EL == EL2 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_Other);
elsif PSTATE.EL == EL3 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_Other);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.3 CPP RCTX, Cache Prefetch Prediction Restriction by Context

The CPP RCTX characteristics are:

Purpose

Cache Prefetch Prediction Restriction by Context applies to all Cache Allocation Resources that predict cache allocations based on information gathered within the target execution context or contexts.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

This instruction applies to all:

- Instruction caches.
- Data caches.
- TLB prefetching hardware used by the executing PE that applies to the supplied context or contexts.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_SPECRES is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CPP RCTX are UNDEFINED.

Attributes

CPP RCTX is a 64-bit System instruction.

Field descriptions

Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 and EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see CPP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

This instruction is treated as a NOP when executed at EL3 and either:

- CPP_RCTX.{NSE, NS} selects a reserved value.
- CPP_RCTX.{NSE, NS} == {1, 0} and CPP_RCTX.EL has a value other than `0b11`.

Otherwise:

Security State.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
`0b00`	EL0.	
`0b01`	EL1	
`0b10`	EL2	FEAT_EL2 is implemented
`0b11`	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
`0b0`	Applies to specified ASID for an EL0 target execution context.
`0b1`	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing CPP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
CPP RCTX, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b111

```
if !(IsFeatureImplemented(FEAT_SPECRES) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.EnRCTX == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.CPPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.EnRCTX == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.CPPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
elsif PSTATE.EL == EL2 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
elsif PSTATE.EL == EL3 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.4 DVP RCTX, Data Value Prediction Restriction by Context

The DVP RCTX characteristics are:

Purpose

Data Value Prediction Restriction by Context applies to all Data Value Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Note

The prediction of the PSTATE.{N,Z,C,V} values is not considered a data value for this purpose.

Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_SPECRES is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DVP RCTX are UNDEFINED.

Attributes

DVP RCTX is a 64-bit System instruction.

Field descriptions

Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see DVP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

This instruction is treated as a NOP when executed at EL3 and either:

- DVP_RCTX.{NSE, NS} selects a reserved value.
- DVP_RCTX.{NSE, NS} == {1, 0} and DVP_RCTX.EL has a value other than `0b11`.

Otherwise:

Security State.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
`0b00`	EL0.	
`0b01`	EL1	
`0b10`	EL2	FEAT_EL2 is implemented
`0b11`	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
`0b0`	Applies to specified ASID for an EL0 target execution context.
`0b1`	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing DVP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DVP RCTX, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_SPECRES) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.EnRCTX == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3.FGTEn == '1') && HFGITR_EL2.DVPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.EnRCTX == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.DVPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
elsif PSTATE.EL == EL2 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
elsif PSTATE.EL == EL3 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7 A64 System instructions for the Branch Record Buffer Extension

This section lists the A64 System instructions for the Branch Record Buffer Extension.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7.1 BRB IALL, Invalidate the Branch Record Buffer

The BRB IALL characteristics are:

Purpose

Invalidates all Branch records in the Branch Record Buffer.

Configuration

This system instruction is present only when FEAT_BRBE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to BRB IALL are UNDEFINED.

Attributes

BRB IALL is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing BRB IALL

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
BRB IALL
```

op0	op1	CRn	CRm	op2
0b01	0b001	0b0111	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_BRBE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3.SBRBE IN {'x0'} && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.nBRBIALL == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE IN {'x0'} && SCR_EL3.NS == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        BRB_IALL();
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3.SBRBE IN {'x0'} && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE IN {'x0'} && SCR_EL3.NS == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        BRB_IALL();
elsif PSTATE.EL == EL3 then
    BRB_IALL();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7.2 BRB INJ, Branch Record Injection into the Branch Record Buffer

The BRB INJ characteristics are:

Purpose

Injects the Branch Record held in BRBINFINJ_EL1, BRBSRCINJ_EL1, and BRBTGTINJ_EL1 into the Branch Record Buffer.

Configuration

This system instruction is present only when FEAT_BRBE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to BRB INJ are UNDEFINED.

Attributes

BRB INJ is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing BRB INJ

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
BRB INJ
```

op0	op1	CRn	CRm	op2
0b01	0b001	0b0111	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_BRBE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3.SBRBE IN {'x0'} && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.nBRBINJ == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE IN {'x0'} && SCR_EL3.NS == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        BRB_INJ();
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3.SBRBE IN {'x0'} && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE IN {'x0'} && SCR_EL3.NS == '1' then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        BRB_INJ();
elsif PSTATE.EL == EL3 then
    BRB_INJ();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.8 A64 System instructions for the Trace Extension

This section lists the A64 System instructions for the Trace Extension.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.8.1 TRCIT, Trace Instrumentation

The TRCIT characteristics are:

Purpose

Generates an instrumentation packet in the trace.

Configuration

This system instruction is present only when FEAT_ITE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TRCIT are UNDEFINED.

Attributes

TRCIT is a 64-bit System instruction.

Field descriptions

VALUE, bits [63:0]

Value to be included in the Instrumentation packet.

Executing TRCIT

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TRCIT <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    AArch64.TRCIT(X[t, 64]);
elsif PSTATE.EL == EL1 then
    AArch64.TRCIT(X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TRCIT(X[t, 64]);
elsif PSTATE.EL == EL3 then
    AArch64.TRCIT(X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9 A64 System instructions for the Guarded Control Stack

This section lists the A64 System instructions for the Guarded Control Stack.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.1 GCSPOPCX, Guarded Control Stack Pop and Compare exception return record

The GCSPOPCX characteristics are:

Purpose

Loads an exception return record from the location indicated by the current Guarded Control Stack Pointer register, compares the values loaded with the current ELR_ELx, SPSR_ELx, and LR, and increments the current Guarded Control Stack Pointer register by the size of a Guarded Control Stack exception return record.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPOPCX are UNDEFINED.

Attributes

GCSPOPCX is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing GCSPOPCX

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`GCSPOPCX`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.nGCSEPP == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL1) then
        GCSPOPCX();
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    elsif GCSEnabled(EL2) then
        GCSPOPCX();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif GCSEnabled(EL3) then
        GCSPOPCX();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.2 GCSPOPM, Guarded Control Stack Pop

The GCSPOPM characteristics are:

Purpose

Loads the 64-bit doubleword that is pointed to by the current Guarded Control Stack Pointer, writes it to the destination register, and increments the current Guarded Control Stack Pointer register by the size of a Guarded Control Stack procedure return record.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPOPM are UNDEFINED.

Attributes

GCSPOPM is a 64-bit System instruction.

Field descriptions

Output, bits [63:0]

Output value for Guarded Control Stack procedure return record.

Executing GCSPOPM

This system instruction is an alias of the SYSL instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
GCSPOPM {<Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif GCSEnabled(PSTATE.EL) then
    X[t, 64] = GCSPOPM();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.3 GCSPOPX, Guarded Control Stack Pop exception return record

The GCSPOPX characteristics are:

Purpose

Loads an exception return record from the location indicated by the current Guarded Control Stack Pointer register, checks that the record is a Guarded Control Stack exception return record, and increments the current Guarded Control Stack Pointer register by the size of a Guarded Control Stack exception return record.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPOPX are UNDEFINED.

Attributes

GCSPOPX is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing GCSPOPX

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`GCSPOPX`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0111	0b110

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if GCSEnabled(EL1) then
        GCSPOPX();
elsif PSTATE.EL == EL2 then
    if GCSEnabled(EL2) then
        GCSPOPX();
elsif PSTATE.EL == EL3 then
    if GCSEnabled(EL3) then
        GCSPOPX();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.4 GCSPUSHM, Guarded Control Stack Push

The GCSPUSHM characteristics are:

Purpose

Decrements the current Guarded Control Stack Pointer register by the size of a Guarded Control Stack procedure return record and stores an entry to the Guarded Control Stack.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPUSHM are UNDEFINED.

Attributes

GCSPUSHM is a 64-bit System instruction.

Field descriptions

Input, bits [63:0]

Input value for Guarded Control Stack procedure return record.

Executing GCSPUSHM

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
GCSPUSHM <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0111	0b000

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if (!EL2Enabled() || HCR_EL2.TGE != '1') && GCSCRE0_EL1.PUSHMEn == '0' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && HCR_EL2.TGE == '1' && GCSCRE0_EL1.PUSHMEn == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL0) then
        GCSPUSHM(X[t, 64]);
elsif PSTATE.EL == EL1 then
    if GCSCR_EL1.PUSHMEn == '0' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        ↪HFGITR_EL2.nGCSPUSHM_EL1 == '0' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL1) then
        GCSPUSHM(X[t, 64]);
elsif PSTATE.EL == EL2 then
    if GCSCR_EL2.PUSHMEn == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL2) then
        GCSPUSHM(X[t, 64]);
elsif PSTATE.EL == EL3 then
    if GCSCR_EL3.PUSHMEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif GCSEnabled(EL3) then
        GCSPUSHM(X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.5 GCSPUSHX, Guarded Control Stack Push exception return record

The GCSPUSHX characteristics are:

Purpose

Decrements the current Guarded Control Stack Pointer register by the size of a Guarded Control Stack exception return record and stores a Guarded Control Stack exception return record to the Guarded Control Stack.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPUSHX are UNDEFINED.

Attributes

GCSPUSHX is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing GCSPUSHX

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`GCSPUSHX`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0111	0b100

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '0' then
        EXLOCKException();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
         ↪HFGITR_EL2.nGCSEPP == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL1) then
        GCSPUSHX();
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '0' then
        EXLOCKException();
    elsif GCSEnabled(EL2) then
        GCSPUSHX();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '0' then
        EXLOCKException();
    elsif GCSEnabled(EL3) then
        GCSPUSHX();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.6 GCSSS1, Guarded Control Stack Switch Stack 1

The GCSSS1 characteristics are:

Purpose

Validates that the stack being switched to contains a Valid cap entry, stores an In-progress cap entry on to the stack that is getting switched to and sets the current Guarded Control Stack Pointer to the stack that is getting switched to.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSSS1 are UNDEFINED.

Attributes

GCSSS1 is a 64-bit System instruction.

Field descriptions

IA, bits [63:0]

Incoming address, for the incoming Guarded Control Stack.

Executing GCSSS1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
GCSSS1 <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif GCSEnabled(PSTATE.EL) then
    GCSSS1(X[t, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.7 GCSSS2, Guarded Control Stack Switch Stack 2

The GCSSS2 characteristics are:

Purpose

Validates that the most recent entry of the Guarded Control Stack that is getting switched to contains an In-progress cap entry, stores a Valid cap entry to the Guarded Control Stack that is getting switched from, and sets Xt to the address of that Valid cap entry.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSSS2 are UNDEFINED.

Attributes

GCSSS2 is a 64-bit System instruction.

Field descriptions

OA, bits [63:0]

Outgoing address, for the outgoing Guarded Control Stack.

Executing GCSSS2

This system instruction is an alias of the SYSL instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
GCSSS2 <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    UNDEFINED;
elsif GCSEnabled(PSTATE.EL) then
    X[t, 64] = GCSSS2();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.10 A64 IMPLEMENTATION DEFINED System instructions

This section lists the A64 IMPLEMENTATION DEFINED System instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.10.1 SYS|SYSL|SYSP S1_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED System instructions

The SYS|SYSL|SYSP S1_<op1><*Cn*><Cm>_<op2> characteristics are:

Purpose

This area of the System instruction encoding space is reserved for IMPLEMENTATION DEFINED System instructions.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SYS|SYSL|SYSP S1_<op1>_<Cn>_<Cm>_<op2> are UNDEFINED.

Attributes

SYS|SYSL|SYSP S1_<op1><*Cn*><Cm>_<op2> is a:

- 128-bit System instruction when FEAT_SYSINSTR128 is implemented.
- 64-bit System instruction otherwise.

Field descriptions

When FEAT_SYSINSTR128 is implemented:

IMPLEMENTATION DEFINED, bits [127:0]

IMPLEMENTATION DEFINED.

Otherwise:

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Executing SYS|SYSL|SYSP S1_<op1>_<Cn>_<Cm>_<op2>

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.TIDCP == '1' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL2 then
    AArch64.ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL3 then
    AArch64.ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
```

```
SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

op0	op1	CRn	CRm	op2
0b01	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.TIDCP == '1' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL2 then
    AArch64.ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL3 then
    AArch64.ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_SYSINSTR128 is implemented SYSP #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.TIDCP == '1' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x14);
        else
            AArch64.SystemAccessTrap(EL1, 0x14);
    elsif ELIsInHost(EL0) && SCTLR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.ImpDefSysInstr128(op0, op1, CRn, CRm, op2, t, t2);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x14);
    else
        AArch64.ImpDefSysInstr128(op0, op1, CRn, CRm, op2, t, t2);
elsif PSTATE.EL == EL2 then
    AArch64.ImpDefSysInstr128(op0, op1, CRn, CRm, op2, t, t2);
elsif PSTATE.EL == EL3 then
    AArch64.ImpDefSysInstr128(op0, op1, CRn, CRm, op2, t, t2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C6
A64 Base Instruction Descriptions

This chapter describes the A64 base instructions.

It contains the following sections:

- About the A64 base instructions.
- Alphabetical list of A64 base instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.1 About the A64 base instructions

Alphabetical list of A64 base instructions gives full descriptions of the A64 instructions that are in the following instruction groups:

- Branch, Exception generation, and System instructions.
- Loads and stores associated with the general-purpose registers.
- Data processing (immediate).
- Data processing (register).

A64 instruction set encoding provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

The rest of this section is general description of the base instructions. It contains the following subsections:

- Register size.
- Use of the PC.
- Use of the stack pointer.
- Condition flags and related instructions.

C6.1.1 Register size

Most data processing, comparison, and conversion instructions that use the general-purpose registers as the source or destination operand have two instruction variants that operate on either a 32-bit or a 64-bit value.

Where a 32-bit instruction form is selected, the following holds:

- The upper 32 bits of the source registers are ignored.
- The upper 32 bits of the destination register are set to zero.
- Right shifts and right rotates inject at bit[31], not at bit[63].
- The Condition flags, where set by the instruction, are computed from the lower 32 bits.

This distinction applies even when the results of a 32-bit instruction form are indistinguishable from the lower 32 bits computed by the equivalent 64-bit instruction form. For example, a 32-bit bitwise `ORR` could be performed using a 64-bit `ORR` and simply ignoring the top 32 bits of the result. However, the A64 instruction set includes separate 32-bit and 64-bit forms of the `ORR` instruction.

As well as distinct sign-extend or zero-extend instructions, the A64 instruction set also provides the ability to extend and shift the final source register of an `ADD`, `SUB`, `ADDS`, `or SUBS` instruction and the index register of a load/store instruction. This enables array index calculations involving a 64-bit array pointer and a 32-bit array index to be implemented efficiently.

The assembly language notation enables the distinct identification of registers holding 32-bit values and registers holding 64-bit values. See Register names and Register indexed addressing.

C6.1.2 Use of the PC

A64 instructions have limited access to the PC. The only instructions that can read the PC are those that generate a PC relative address:

- ADR and ADRP.
- The Load register (literal) instruction class.
- Direct branches that use an immediate offset.
- The unconditional branch with link instructions, BL and BLR, that use the PC to create the return link address.

Only explicit control flow instructions can modify the PC:

- Conditional and unconditional branch and return instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Exception generation and exception return instructions.

For more details of instructions that can modify the PC, see Branches, Exception generating, and System instructions.

C6.1.3 Use of the stack pointer

A64 instructions can use the stack pointer only in a limited number of cases:

- Load/store instructions use the current stack pointer as the base address:
  - When stack alignment checking is enabled by system software and the base register is SP, the current stack pointer must be initially quadword aligned, That is, it must be aligned to 16 bytes. Misalignment generates an SP alignment fault. See SP alignment checking for more information.
- Add and subtract data processing instructions in their immediate and extended register forms, use the current stack pointer as a source register or the destination register or both.
- Logical data processing instructions in their immediate form use the current stack pointer as the destination register.

C6.1.4 Condition flags and related instructions

The A64 base instructions that use the Condition flags as an input are:

- Conditional branch. The conditional branch instruction is `B.cond`.
- Add or subtract with carry. These instruction types include instructions to perform multi-precision arithmetic and calculate checksums. The add or subtract with carry instructions are `ADC`, `ADCS`, `SBC`, and `SBCS`, or an architectural alias for these instructions.
- Conditional select with increment, negate, or invert. This instruction type conditionally selects between one source register and a second, incremented, negated, inverted, or unmodified source register. The conditional select with increment, negate, or invert instructions are `CSINC`, `CSINV`, and `CSNEG`.

  These instructions also implement:
  - Conditional select or move. The Condition flags select one of two source registers as the destination register. Short conditional sequences can be replaced by unconditional instructions followed by a conditional select, `CSEL`.
  - Conditional set. Conditionally selects between 0 and 1, or 0 and -1. This can be used to convert the Condition flags to a Boolean value or mask in a general-purpose register, for example. These instructions include `CSET` and `CSETM`.
- Conditional compare. This instruction type sets the Condition flags to the result of a comparison if the original condition is true, otherwise it sets the Condition flags to an immediate value. It permits the flattening of nested conditional expressions without using conditional branches or performing Boolean arithmetic within the general-purpose registers. The conditional compare instructions are `CCMP` and `CCMN`.

The A64 base instructions that update the Condition flags as an output are:

- Flag-setting data processing instructions, such as `ADCS`, `ADDS`, `ANDS`, `BICS`, `RMIF`, `SBCS`, `SETF8`,`SETF16`, and `SUBS`, and the aliases `CMN`, `CMP`, and `TST`.
- Conditional compare instructions such as `CCMN`, `CCMP`.
- The random number generation instructions `MRS RNDR` and `MRS RNDRRS`, see Effect of random number generation instructions on Condition flags.

The A64 base instructions that manipulate the Condition flags are:

- The flag manipulation instruction `CFINV`, which inverts the value of the Carry flag.
- If FEAT_FlagM2 is implemented, the base instructions `AXFLAG` and `XAFLAG`. These instructions convert between the Arm floating point comparison PSTATE condition flag format and an alternative format shown in Table C6-1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C6-1 Relationship between ARM format and alternative format PSTATE condition flags

	ARM format				Alternative format			
Result	N	Z	C	V	N	Z	C	V
Greater than	0	0	1	0	0	0	1	0
Less than	1	0	0	0	0	0	0	0
Equal	0	1	1	0	0	1	1	0
Unordered	0	0	1	1	0	1	0	0

The flags can be directly accessed for a read/write using the NZCV.

The A64 base instructions also include conditional branch instructions that do not manipulate the Condition flags:

- Compare and branch if a register is zero or nonzero, `CBZ` and `CBNZ`.
- Test a single bit in a register and branch if the bit is zero or nonzero, `TBZ` and `TBNZ`.
- Compare and branch instructions, `CB<cc>` (immediate), `CB<cc>` (register), `CBB<cc>`, and `CBH<cc>`.

C6.1.4.1 Effect of random number generation instructions on Condition flags

If FEAT_RNG is implemented, then:

- When a valid random number is returned, the PSTATE.NZCV flags are set to `0b0000`.
- If the random number hardware is not capable of returning a random number in a reasonable period of time, the PSTATE.NZCV flags are set to `0b0100`, and the random number generation instructions return the value 0.

Note

The definition of "reasonable period of time" is IMPLEMENTATION DEFINED. The expectation is that software might use this as an opportunity to reschedule or run a different routine, perhaps after a small number of retries have failed to return a valid value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2 Alphabetical list of A64 base instructions

This section lists every instruction in the base category of the A64 instruction set. For details of the format used, see Understanding the A64 instruction descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.1 ABS

Absolute value

This instruction computes the absolute value of the signed integer value in the source register, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28							21	20					16	15						10	9				5	4				0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0		0	0	1	0	0	0		Rn					Rd				
		S									opcode2									opcode													

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ABS  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ABS  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer datasize = 32 << UInt(sf);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = Abs(SInt(operand1));
X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.2 ADC

Add with carry

This instruction adds two register values and the Carry flag value, and writes the result to the destination register.

31	30	29	28	21	20	16	15	10	9	5	4	0
sf	0	0	1 1 0 1 0 0 0 0		Rm		0 0 0 0 0 0		Rn		Rd	
	op	S										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
bits(datasize) result;

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

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.3 ADCS

Add with carry, setting flags

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

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	0	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0		Rn					Rd				
	op			S																												

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

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

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.4 ADD (extended register)

Add extended and scaled register

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

31	30	29	28	24	23	22	21	20	16	15	13	12	10	9	5	4	0
sf	0	0	0 1 0 1 1		0	0	1	Rm		option		imm3		Rn		Rd	

op S opt

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
constant integer datasize = 32 << UInt(sf);
constant ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rd” or “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

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

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

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

<amount>

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

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

option	<R>
00x	W

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<m>

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

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.5 ADD (immediate)

Add immediate value

This instruction adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP).

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

constant bits(24) imm = if sh == '0' then Zeros(12):imm12 else imm12:Zeros(12);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
MOV (to/from SP)	sh == '0' && imm12 == '000000000000' && (Rd == '11111' || Rn == '11111')

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ZeroExtend(imm, datasize);
bits(datasize) result;
(result, -) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.6 ADD (shifted register)

Add optionally-shifted register

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28	24	23 22	21	20	16	15	10	9	5	4	0
sf	0	0	0 1 0 1 1		shift	0	Rm		imm6		Rn		Rd	
	op	S												

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

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

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

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

(result, -) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.7 ADDG

Add with tag

This instruction 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.

Integer

(FEAT_MTE)

Encoding

```
ADDG  <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant bits(4) tag_offset = imm4;
constant bits(64) offset = LSL(ZeroExtend(imm6, 64), LOG2_TAG_GRANULE);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<uimm6>

Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the “imm6” field.

<uimm4>

Is an unsigned immediate, in the range 0 to 15, encoded in the “imm4” field.

Operation

```
constant bits(64) operand1 = if n == 31 then SP[64] else X[n, 64];
constant bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
constant bits(16) exclude = GCR_EL1.Exclude;
constant bits(4) rtag = AArch64.ChooseNonExcludedTagOrZero(start_tag, tag_offset, exclude);
bits(64) result;

(result, -) = AddWithCarry(operand1, offset, '0');

result = AArch64.AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[64] = result;
else
    X[d, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.8 ADDPT

Add checked pointer

This instruction adds a base address register value and an optionally-shifted register value, and writes the result to the destination register. The optionally-shifted register value is treated as the offset.

If the operation would have generated a result where the most significant 8 bits of the result register differ from the most significant 8 bits of the base register, then the result is modified such that it is likely to be non-canonical when used as an address.

Integer

(FEAT_CPA)

31	30	29	28	27 25	24 21	20 16	15 13	12 10	9 5	4 0
1	0	0	1	1 0 1	0 0 0 0	Rm	0 0 1	imm3	Rn	Rd
sf	op	S								

Encoding

```
ADDPT  <Xd|SP>, <Xn|SP>, <Xm>{, LSL #<amount>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CPA) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the general-purpose destination register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first general-purpose source register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the left shift amount, in the range 0 to 7, defaulting to 0, encoded in the “imm3” field.

Operation

```
bits(64) result;
constant bits(64) base = if n == 31 then SP[64] else X[n, 64];
constant bits(64) offset = LSL(X[m, 64], shift);

result = base + offset;
result = PointerAddCheck(result, base);

if d == 31 then
    SP[64] = result;
else
    X[d, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.9 ADDS (extended register)

Add extended and scaled register, setting flags

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

This instruction is used by the alias CMN (extended register).

31	30	29	28				24	23	22	21	20		16	15		13	12		10	9		5	4		0
sf	0	1	0	1	0	1	1	0	0	1	Rm			option			imm3			Rn			Rd		
op		S						opt																	

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
constant integer datasize = 32 << UInt(sf);
constant ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

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

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

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

<amount>

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

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<m>

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

Alias Conditions

Alias	Is preferred when
CMN (extended register)	Rd == '11111'

Operation

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

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.10 ADDS (immediate)

Add immediate value, setting flags

This instruction adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (immediate).

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

constant bits(24) imm = if sh == '0' then Zeros(12):imm12 else imm12:Zeros(12);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
CMN (immediate)	Rd == '11111'

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ZeroExtend(imm, datasize);
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.11 ADDS (shifted register)

Add optionally-shifted register, setting flags

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register).

31	30	29	28				24	23 22	21	20	16	15	10	9	5	4	0
sf	0	1	0	1	0	1	1	shift	0	Rm		imm6		Rn		Rd	
	op																
		S															

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

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

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMN (shifted register)	Rd == '11111'

Operation

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

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.12 ADR

Form PC-relative address

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

31	30 29	28 26	25 24	23 5	4 0
0	immlo	1 0 0	0 0	immhi	Rd
op					

Encoding

```
ADR  <Xd>, <label>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant bits(64) imm = SignExtend(immhi:immlo, 64);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<label>

Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in “immhi:immlo”.

Operation

```
X[d, 64] = PC64 + imm;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.13 ADRP

Form PC-relative address to 4KB page

This instruction 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.

31	30 29	28 26	25 24	23 5	4 0
1	immlo	1 0 0	0 0	immhi	Rd
op					

Encoding

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

Decode for this encoding

```
constant integer d = UInt(Rd);
constant bits(64) imm = SignExtend(immhi:immlo:Zeros(12), 64);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<label>

Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as “immhi:immlo” times 4096.

Operation

```
constant bits(64) base = PC64<63:12>:Zeros(12);
X[d, 64] = base + imm;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.14 AND (immediate)

Bitwise AND (immediate)

This instruction performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 0 0	N	immr	imms	Rn	Rd
	opc						

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”.

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

constant bits(datasize) result = operand1 AND operand2;

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.15 AND (shifted register)

Bitwise AND (shifted register)

This instruction performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 0 (opc)	0 1 0 1 0	shift	0 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

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

X[d, datasize] = operand1 AND operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.16 ANDS (immediate)

Bitwise AND (immediate), setting flags

This instruction performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (immediate).

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 1 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
TST (immediate)	Rd == '11111'

Operation

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

constant bits(datasize) result = operand1 AND operand2;

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.17 ANDS (shifted register)

Bitwise AND (shifted register), setting flags

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

This instruction is used by the alias TST (shifted register).

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 1 (opc)	0 1 0 1 0	shift	0 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
TST (shifted register)	Rd == '11111'

Operation

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

constant bits(datasize) result = operand1 AND operand2;
X[d, datasize] = result;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.18 APAS

Associate physical address space

This instruction associates a physical address space with a memory-mapped location that is protected by a memory-side physical address space filter.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_RME_GPC3)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	1 1 0	0 1 1 1	0 0 0 0	0 0 0	Rt
			L	op1		CRn	CRm	op2	

Encoding

```
APAS  <Xt>
```

is equivalent to

```
SYS  #6, C7, C0, #0, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.19 ASR (register)

Arithmetic shift right (register)

This instruction 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 value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of ASRV. This means:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0		1	0	Rn					Rd				
		S																				op2										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ASR  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
ASRV  <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ASR  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
ASRV  <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of ASRV gives the operational pseudocode for this instruction.

Operational Information

The description of ASRV gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.20 ASR (immediate)

Arithmetic shift right (immediate)

This instruction 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.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 0	N	immr	x 1 1 1 1 1	Rn	Rd
	opc				imms		

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms == 011111)`

```
ASR  <Wd>, <Wn>, #<shift>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1 && imms == 111111)`

```
ASR  <Xd>, <Xn>, #<shift>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<shift>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, encoded in the “immr” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.21 ASRV

Arithmetic shift right variable

This instruction 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 value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ASR (register).

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0		1	0	Rn					Rd			
		S																				op2									

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ASRV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ASRV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
ASR (register)	Unconditionally

Operation

```
constant bits(datasize) operand2 = X[m, datasize];

X[d, datasize] = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.22 AT

Address translate

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding

```
AT  <at_op>, <Xt>
```

is equivalent to

```
SYS  #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_AT`.

Assembler Symbols

<at_op>

Is an AT operation name, as listed for the AT system instruction group, encoded in “op1:CRm:op2”:

op1	CRm	op2	<at_op>	Architectural Feature
000	1000	000	S1E1R	-
000	1000	001	S1E1W	-
000	1000	010	S1E0R	-
000	1000	011	S1E0W	-
000	1001	000	S1E1RP	FEAT_PAN2
000	1001	001	S1E1WP	FEAT_PAN2
000	1001	010	S1E1A	FEAT_ATS1A
100	1000	000	S1E2R	-
100	1000	001	S1E2W	-
100	1000	100	S12E1R	-
100	1000	101	S12E1W	-
100	1000	110	S12E0R	-
100	1000	111	S12E0W	-
100	1001	010	S1E2A	FEAT_ATS1A
110	1000	000	S1E3R	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRm	op2	<at_op>	Architectural Feature
110	1000	001	S1E3W	-
110	1001	010	S1E3A	FEAT_ATS1A

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.23 AUTDA, AUTDZA

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. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer

(FEAT_PAuth)

Encoding for the AUTDA variant

Applies when `(Z == 0)`

```
AUTDA  <Xd>, <Xn|SP>
```

Encoding for the AUTDZA variant

Applies when `(Z == 1 && Rn == 11111)`

```
AUTDZA  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF);
constant boolean auth_combined = FALSE;

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant boolean source_is_sp = Z == '0' && n == 31;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    X[d, 64] = AuthDA(X[d, 64], SP[64], auth_combined);
else
    X[d, 64] = AuthDA(X[d, 64], X[n, 64], auth_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.24 AUTDB, AUTDZB

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. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer

(FEAT_PAuth)

31	30	29	28							21	20				16	15	14	13	12		10	9		5	4		0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	1	1	1	Rn			Rd		
sf			S												opcode2												

Encoding for the AUTDB variant

Applies when `(Z == 0)`

```
AUTDB  <Xd>, <Xn|SP>
```

Encoding for the AUTDZB variant

Applies when `(Z == 1 && Rn == 11111)`

```
AUTDZB  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF);
constant boolean auth_combined = FALSE;

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant boolean source_is_sp = Z == '0' && n == 31;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    X[d, 64] = AuthDB(X[d, 64], SP[64], auth_combined);
else
    X[d, 64] = AuthDB(X[d, 64], X[n, 64], auth_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.25 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate instruction address, using key A

This instruction authenticates an instruction address, using a modifier and key A.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

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 FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `AUTIA1716` and `AUTIASP` include a second modifier that is:

- In X15, for `AUTIA1716`.
- In X16, for `AUTIASP`.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

Encoding for the AUTIA variant

Applies when `(Z == 0)`

```
AUTIA  <Xd>, <Xn|SP>
```

Encoding for the AUTIZA variant

Applies when `(Z == 1 && Rn == 11111)`

```
AUTIZA  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF);
constant boolean autia1716 = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant boolean auth_combined = FALSE;

constant boolean source_is_sp = Z == '0' && n == 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System

(FEAT_PAuth)

31																				12	11			8	7			5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0		0	0	x	1	1	0	x		1	1	1	1	1
																					CRm					op2							

Encoding for the AUTIA1716 variant

Applies when `(CRm == 0001 && op2 == 100)`

```
AUTIA1716
```

Encoding for the AUTIASP variant

Applies when `(CRm == 0011 && op2 == 101)`

```
AUTIASP
```

Encoding for the AUTIAZ variant

Applies when `(CRm == 0011 && op2 == 100)`

```
AUTIAZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
integer d;
integer n;
boolean source_is_sp = FALSE;
boolean autia1716 = FALSE;
constant boolean auth_combined = FALSE;

case CRm:op2 of
    when '0011 100' // AUTIAZ
        d = 30;
        n = 31;
    when '0011 101' // AUTIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 100' // AUTIA1716
        d = 17;
        n = 16;
        autia1716 = TRUE;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if source_is_sp then
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
        X[d, 64] = AuthIA2(X[d, 64], SP[64], X[16, 64], auth_combined);
    else
        X[d, 64] = AuthIA(X[d, 64], SP[64], auth_combined);
else
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' && autia1716 then
        X[d, 64] = AuthIA2(X[d, 64], X[n, 64], X[15, 64], auth_combined);
    else
        X[d, 64] = AuthIA(X[d, 64], X[n, 64], auth_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.26 AUTIA171615

Authenticate instruction address, using key A

This instruction authenticates an instruction address, using two modifiers and key A. The address is in X17. The 64-bit value of modifier1 is the value in X16. The 64-bit value of modifier2 is the value in X15.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer

(FEAT_PAuth_LR)

Encoding

```
AUTIA171615
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);
```

Operation

```
constant boolean is_combined = FALSE;
X[17, 64] = AuthIA2(X[17, 64], X[16, 64], X[15, 64], is_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.27 AUTIASPPC

Authenticate return address using key A, using an immediate offset

This instruction authenticates an instruction address, using two modifiers and key A.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is in X30.

The first modifier is in SP.

The second modifier is the address of a program label.

Integer

(FEAT_PAuth_LR)

31	30 29	28 26	25 23	22 21	20 5	4 0
1	1 1	1 0 0	1 1 1	0 0	imm16	1 1 1 1 1
sf				opc		Rd

Encoding

```
AUTIASPPC  <label>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant integer d = 30;
constant bits(64) offset = ZeroExtend(imm16:'00', 64);
constant boolean auth_combined = FALSE;
```

Assembler Symbols

<label>

Is the program label whose address is to be calculated. Its negative offset from the address of this instruction, a multiple of 4 in the range -262140 to 0, is encoded as an unsigned value in the “imm16” field as <label>/4.

Operation

```
constant bits(64) pac_addr = PC64 - offset;

X[d, 64] = AuthIA2(X[d, 64], SP[64], pac_addr, auth_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.28 AUTIASPPCR

Authenticate return address using key A, using a register

This instruction authenticates an instruction address, using two modifiers and key A.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is in X30.

The first modifier is in SP.

The second modifier is in the general-purpose register that is specified by `<Xn>`.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	1	0	0	Rn					1	1	1	1	0
sf		S									opcode2					opcode											Rd				

Encoding

```
AUTIASPPCR  <Xn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant integer d = 30;
constant integer n = UInt(Rn);
constant boolean auth_combined = FALSE;
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
X[d, 64] = AuthIA2(X[d, 64], SP[64], X[n, 64], auth_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.29 AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate instruction address, using key B

This instruction authenticates an instruction address, using a modifier and key B.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

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 FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `AUTIB1716` and `AUTIBSP` include a second modifier that is:

- In X15, for `AUTIB1716`.
- In X16, for `AUTIBSP`.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

Encoding for the AUTIB variant

Applies when `(Z == 0)`

```
AUTIB  <Xd>, <Xn|SP>
```

Encoding for the AUTIZB variant

Applies when `(Z == 1 && Rn == 11111)`

```
AUTIZB  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF);
constant boolean autib1716 = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant boolean auth_combined = FALSE;

constant boolean source_is_sp = Z == '0' && n == 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System

(FEAT_PAuth)

31																				12	11			8	7			5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0		0	0	x	1	1	1	x		1	1	1	1	1
																					CRm						op2						

Encoding for the AUTIB1716 variant

Applies when `(CRm == 0001 && op2 == 110)`

```
AUTIB1716
```

Encoding for the AUTIBSP variant

Applies when `(CRm == 0011 && op2 == 111)`

```
AUTIBSP
```

Encoding for the AUTIBZ variant

Applies when `(CRm == 0011 && op2 == 110)`

```
AUTIBZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
integer d;
integer n;
boolean source_is_sp = FALSE;
boolean autib1716 = FALSE;
constant boolean auth_combined = FALSE;

case CRm:op2 of
    when '0011 110' // AUTIBZ
        d = 30;
        n = 31;
    when '0011 111' // AUTIBSP
        d = 30;
        source_is_sp = TRUE;
    when '0001 110' // AUTIB1716
        d = 17;
        n = 16;
        autib1716 = TRUE;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if source_is_sp then
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
        X[d, 64] = AuthIB2(X[d, 64], SP[64], X[16, 64], auth_combined);
    else
        X[d, 64] = AuthIB(X[d, 64], SP[64], auth_combined);
else
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' && autib1716 then
        X[d, 64] = AuthIB2(X[d, 64], X[n, 64], X[15, 64], auth_combined);
    else
        X[d, 64] = AuthIB(X[d, 64], X[n, 64], auth_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.30 AUTIB171615

Authenticate instruction address, using key B

This instruction authenticates an instruction address, using two modifiers and key B. The address is in X17. The 64-bit value of modifier1 is the value in X16. The 64-bit value of modifier2 is the value in X15.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer

(FEAT_PAuth_LR)

Encoding

```
AUTIB171615
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);
```

Operation

```
constant boolean is_combined = FALSE;
X[17, 64] = AuthIB2(X[17, 64], X[16, 64], X[15, 64], is_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.31 AUTIBSPPC

Authenticate return address using key B, using an immediate offset

This instruction authenticates an instruction address, using two modifiers and key B.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is in X30.

The first modifier is in SP.

The second modifier is the address of a program label.

Integer

(FEAT_PAuth_LR)

31	30 29	28 26	25 23	22 21	20 5	4 0
1	1 1	1 0 0	1 1 1	0 1	imm16	1 1 1 1 1
sf				opc		Rd

Encoding

```
AUTIBSPPC  <label>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant integer d = 30;
constant bits(64) offset = ZeroExtend(imm16:'00', 64);
constant boolean auth_combined = FALSE;
```

Assembler Symbols

<label>

Is the program label whose address is to be calculated. Its negative offset from the address of this instruction, a multiple of 4 in the range -262140 to 0, is encoded as an unsigned value in the “imm16” field as <label>/4.

Operation

```
constant bits(64) pac_addr = PC64 - offset;

X[d, 64] = AuthIB2(X[d, 64], SP[64], pac_addr, auth_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.32 AUTIBSPPCR

Authenticate return address using key B, using a register

This instruction authenticates an instruction address, using two modifiers and key B.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is in X30.

The first modifier is in SP.

The second modifier is in the general-purpose register that is specified by `<Xn>`.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	1	0	1	Rn					1	1	1	1	0
sf		S									opcode2					opcode											Rd				

Encoding

```
AUTIBSPPCR  <Xn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant integer d = 30;
constant integer n = UInt(Rn);
constant boolean auth_combined = FALSE;
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
X[d, 64] = AuthIB2(X[d, 64], SP[64], X[n, 64], auth_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.33 AXFLAG

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.

System

(FEAT_FlagM2)

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	0 0 0	0 1 0 0	(0) (0) (0) (0)	0 1 0	1 1 1 1 1
			op1		CRm	op2	Rt

Encoding

```
AXFLAG
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM2) then EndOfDecode(Decode_UNDEF);
```

Operation

```
constant bit z = PSTATE.Z OR PSTATE.V;
constant bit c = PSTATE.C AND NOT(PSTATE.V);

PSTATE.<N,Z,C,V> = '0' : z : c : '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.34 B.cond

Branch conditionally

This instruction branches conditionally to a label at a PC-relative offset. This instruction provides a hint that this is not a subroutine call or return.

Encoding

```
B.<cond>  <label>
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm19:'00', 64);
constant bits(4) condition = cond;
```

Assembler Symbols

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant boolean branch_conditional = TRUE;
if ConditionHolds(condition) then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.35 B

Branch

This instruction branches unconditionally to a label at a PC-relative offset. This instruction provides a hint that this is not a subroutine call or return.

31	30	29	28		26	25 … 0
0	0	0	1	0	1	imm26
op						

Encoding

```
B  <label>
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

<label>

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as “imm26” times 4.

Operation

```
constant boolean branch_conditional = FALSE;
BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.36 BC.cond

Branch consistent conditionally

This instruction branches conditionally to a label at a PC-relative offset, with a hint that this branch is very unlikely to change direction. This instruction provides a hint that this is not a subroutine call or return.

19-bit signed PC-relative branch offset

(FEAT_HBC)

Encoding

```
BC.<cond>  <label>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_HBC) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm19:'00', 64);
constant bits(4) condition = cond;
```

Assembler Symbols

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Operation

```
constant boolean branch_conditional = TRUE;
if ConditionHolds(condition) then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.37 BFC

Bitfield clear

This instruction sets a bitfield of `<width>` bits at bit position `<lsb>` of the destination register to zero, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Leaving other bits unchanged

(FEAT_ASMv8p2)

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 1 (opc)	1 0 0 1 1 0	N	immr	imms	1 1 1 1 1 (Rn)	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
BFC  <Wd>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, WZR, #(-<lsb>  MOD  32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
BFC  <Xd>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, XZR, #(-<lsb>  MOD  64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

The description of BFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.38 BFI

Bitfield insert

This instruction copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 1 (opc)	1 0 0 1 1 0	N	immr	imms	!=11111 (Rn)	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
BFI  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, <Wn>, #(-<lsb>  MOD  32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
BFI  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, <Xn>, #(-<lsb>  MOD  64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

The description of BFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.39 BFM

Bitfield move

This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If `<imms>` is greater than or equal to `<immr>`, this copies a bitfield of (`<imms>`-`<immr>`+1) bits starting from bit position `<immr>` in the source register to the least significant bits of the destination register.

If `<imms>` is less than `<immr>`, this copies a bitfield of (`<imms>`+1) bits from the least significant bits of the source register to bit position (regsize-`<immr>`) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the other bits of the destination register remain unchanged.

This instruction is used by the aliases BFC, BFI, and BFXIL.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 1 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
BFM  <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
BFM  <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF);
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer s = UInt(imms);
constant integer r = UInt(immr);

bits(datasize) wmask;
bits(datasize) tmask;
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<immr>

For the “32-bit” variant: is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imms>

For the “32-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
BFC	`Rn == '11111' && UInt(imms) < UInt(immr)`
BFI	`Rn != '11111' && UInt(imms) < UInt(immr)`
BFXIL	`UInt(imms) >= UInt(immr)`

Operation

```
constant bits(datasize) dst = X[d, datasize];
constant bits(datasize) src = X[n, datasize];

// Perform bitfield move on low bits
constant bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, r) AND wmask);

// Combine extension bits and result bits
X[d, datasize] = (dst AND NOT(tmask)) OR (bot AND tmask);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.40 BFXIL

Bitfield extract and insert at low end

This instruction copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 1 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
BFXIL  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
BFXIL  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

The description of BFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.41 BIC (shifted register)

Bitwise bit clear (shifted register)

This instruction performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 0	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

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

X[d, datasize] = operand1 AND NOT(operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.42 BICS (shifted register)

Bitwise bit clear (shifted register), setting flags

This instruction performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 1 (opc)	0 1 0 1 0	shift	1 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

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

constant bits(datasize) result = operand1 AND NOT(operand2);
X[d, datasize] = result;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.43 BL

Branch with link

This instruction branches to a PC-relative offset, setting register X30 to PC+4. This instruction provides a hint that this is a subroutine call.

31	30 29	28 26	25 0
1	0 0	1 0 1	imm26
op			

Encoding

```
BL  <label>
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm26:'00', 64);
constant integer d = 30;
```

Assembler Symbols

<label>

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as “imm26” times 4.

Operation

```
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64 + 4);
X[d, 64] = PC64 + 4;

constant boolean branch_conditional = FALSE;
BranchTo(PC64 + offset, BranchType_DIRCALL, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.44 BLR

Branch with link to register

This instruction calls a subroutine at an address in a register, setting register X30 to PC+4. This instruction provides a hint that this is a subroutine call.

Encoding

```
BLR  <Xn>
```

Decode for this encoding

```
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

Operation

```
constant bits(64) target = X[n, 64];
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64 + 4);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';

X[30, 64] = PC64 + 4;

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_INDCALL, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.45 BLRAA, BLRAAZ, BLRAB, BLRABZ

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`. Key B is used for `BLRAB` and `BLRABZ`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to the general-purpose register.

This instruction provides a hint that this is a subroutine call.

Integer

(FEAT_PAuth)

Encoding for the Key A, register modifier variant

Applies when `(Z == 1 && M == 0)`

```
BLRAA  <Xn>, <Xm|SP>
```

Encoding for the Key A, zero modifier variant

Applies when `(Z == 0 && M == 0 && Rm == 11111)`

```
BLRAAZ  <Xn>
```

Encoding for the Key B, register modifier variant

Applies when `(Z == 1 && M == 1)`

```
BLRAB  <Xn>, <Xm|SP>
```

Encoding for the Key B, zero modifier variant

Applies when `(Z == 0 && M == 1 && Rm == 11111)`

```
BLRABZ  <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '0' && Rm != '11111' then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean use_key_a = (M == '0');
constant boolean source_is_sp = ((Z == '1') && (m == 31));
constant boolean auth_then_branch = TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the “Rm” field.

Operation

```
bits(64) target = X[n, 64];
constant bits(64) modifier = if source_is_sp then SP[64] else X[m, 64];

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64 + 4);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';

X[30, 64] = PC64 + 4;

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_INDCALL, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.46 BR

Branch to register

This instruction branches unconditionally to an address in a register. This instruction provides a hint that this is not a subroutine call or return.

Encoding

```
BR  <Xn>
```

Decode for this encoding

```
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

Operation

```
constant bits(64) target = X[n, 64];
// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_INDIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.47 BRAA, BRAAZ, BRAB, BRABZ

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`. Key B is used for `BRAB` and `BRABZ`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to the general-purpose register.

This instruction provides a hint that this is not a subroutine call or return.

Integer

(FEAT_PAuth)

Encoding for the Key A, register modifier variant

Applies when `(Z == 1 && M == 0)`

```
BRAA  <Xn>, <Xm|SP>
```

Encoding for the Key A, zero modifier variant

Applies when `(Z == 0 && M == 0 && Rm == 11111)`

```
BRAAZ  <Xn>
```

Encoding for the Key B, register modifier variant

Applies when `(Z == 1 && M == 1)`

```
BRAB  <Xn>, <Xm|SP>
```

Encoding for the Key B, zero modifier variant

Applies when `(Z == 0 && M == 1 && Rm == 11111)`

```
BRABZ  <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '0' && Rm != '11111' then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean use_key_a = (M == '0');
constant boolean source_is_sp = ((Z == '1') && (m == 31));
constant boolean auth_then_branch = TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the “Rm” field.

Operation

```
bits(64) target = X[n, 64];
constant bits(64) modifier = if source_is_sp then SP[64] else X[m, 64];

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);
// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_INDIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.48 BRB

Branch record buffer

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_BRBE)

Encoding

```
BRB  <brb_op>
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp('001', '0111', '0010', op2) == Sys_BRB`.

Assembler Symbols

<brb_op>

Is a BRB operation name, as listed for the BRB system instruction group, encoded in “op2”:

op2	<brb_op>	Architectural Feature
100	IALL	FEAT_BRBE
101	INJ	FEAT_BRBE

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.49 BRK

Breakpoint instruction

This 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.

Within a guarded memory region, while PSTATE.BTYPE != `0b00`, a `BRK` instruction will not generate a Branch Target exception and will generate a Breakpoint Instruction exception as normal. For more information, see PSTATE.BTYPE.

31–29	28–26	25–24	23–21	20–5	4–2	1–0
1 1 0	1 0 1	0 0	0 0 1	imm16	0 0 0	0 0
			opc		op2	LL

Encoding

```
BRK  #<imm>
```

Decode for this encoding

```
constant bits(16) comment = imm16;
if IsFeatureImplemented(FEAT_BTI) then
    SetBTypeCompatible(TRUE);
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
AArch64.SoftwareBreakpoint(comment);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.50 BTI

Branch target identification

This 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`. Within a guarded memory region, while PSTATE.BTYPE != `0b00`, a `BTI` instruction compatible with the current value of PSTATE.BTYPE will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region. For more information, see PSTATE.BTYPE.

The operand `<targets>` passed to a `BTI` instruction determines the values of PSTATE.BTYPE that the `BTI` instruction is compatible with.

System

(FEAT_BTI)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 1 0 0 (CRm)	x x 0 (op2)	1 1 1 1 1

Encoding

```
BTI  {<targets>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BTI) then EndOfDecode(Decode_NOP);

// Check branch target compatibility between BTI instruction and PSTATE.BTYPE
SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
```

Assembler Symbols

<targets>

Is the type of indirection, encoded in “op2<2:1>”:

op2<2:1>	<targets>
00	[absent]
01	c
10	j
11	jc

Operation

```
SetBTypeNext('00');
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.51 CAS, CASA, CASAL, CASL

Compare and swap word or doubleword in memory

This instruction 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 comparison is not equal, the architecture permits writing the value read from the location 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.

- If the destination register is not one of `WZR` or `XZR`, `CASA` and `CASAL` load from memory with acquire semantics.
  - `CASL` and `CASAL` store to memory with release semantics.
  - `CAS` has neither acquire nor release semantics.

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.

For a `CAS` or `CASA` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CAS`, `CASA`, `CASAL`, or `CASL` access to the specified location is likely to occur in the near future. The memory system can respond by taking actions that are expected to enable the subsequent `CAS`, `CASA`, `CASAL`, or `CASL` access to succeed when it does occur.

A code sequence starting with a `CAS` or `CASA` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CAS`, `CASA`, `CASAL`, or `CASL` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CAS` or `CASA` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CAS` or `CASA` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CAS` or `CASA` either fails its compare or writes the same value back to memory.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSE)

31	30	29						23	22	21	20			16	15	14				10	9			5	4			0
1	x	0	0	1	0	0	0	1	L	1	Rs				o0	1	1	1	1	1	Rn				Rt			
size																Rt2												

Encoding for the 32-bit CAS variant

Applies when `(size == 10 && L == 0 && o0 == 0)`

```
CAS  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASA variant

Applies when `(size == 10 && L == 1 && o0 == 0)`

```
CASA  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 32-bit CASAL variant

Applies when `(size == 10 && L == 1 && o0 == 1)`

```
CASAL  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASL variant

Applies when `(size == 10 && L == 0 && o0 == 1)`

```
CASL  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CAS variant

Applies when `(size == 11 && L == 0 && o0 == 0)`

```
CAS  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASA variant

Applies when `(size == 11 && L == 1 && o0 == 0)`

```
CASA  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASAL variant

Applies when `(size == 11 && L == 1 && o0 == 1)`

```
CASAL  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASL variant

Applies when `(size == 11 && L == 0 && o0 == 1)`

```
CASL  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean acquire = L == '1' && t != 31;
constant boolean release = o0 == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

Operation

```
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          tagchecked, privileged, t, s);
comparevalue = X[s, datasize];
newvalue = X[t, datasize];

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) data = MemAtomic(address, comparevalue, newvalue, accdesc);
X[s, regsize] = ZeroExtend(data, regsize);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.52 CASB, CASAB, CASALB, CASLB

Compare and swap byte in memory

This instruction 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 comparison is not equal, the architecture permits writing the value read from the location 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.

- If the destination register is not one of `WZR` or `XZR`, `CASAB` and `CASALB` load from memory with acquire semantics.
  - `CASLB` and `CASALB` store to memory with release semantics.
  - `CASB` has neither acquire nor release semantics.

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.

For a `CASB` or `CASAB` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` access to the specified location is likely to occur in the near future. The memory system can respond by taking actions that are expected to enable the subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` access to succeed when it does occur.

A code sequence starting with a `CASB` or `CASAB` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CASB` or `CASAB` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CASB` or `CASAB` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CASB` or `CASAB` either fails its compare or writes the same value back to memory.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSE)

31 30	29 … 23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0 0	0 0 1 0 0 0 1	L	1	Rs	o0	1 1 1 1 1	Rn	Rt
size						Rt2		

Encoding for the CASB variant

Applies when `(L == 0 && o0 == 0)`

```
CASB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASAB variant

Applies when `(L == 1 && o0 == 0)`

```
CASAB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the CASALB variant

Applies when `(L == 1 && o0 == 1)`

```
CASALB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASLB variant

Applies when `(L == 0 && o0 == 1)`

```
CASLB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = L == '1' && t != 31;
constant boolean release = o0 == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          tagchecked, privileged, t, s);
comparevalue = X[s, 8];
newvalue = X[t, 8];

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) data = MemAtomic(address, comparevalue, newvalue, accdesc);
X[s, 32] = ZeroExtend(data, 32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.53 CASH, CASAH, CASALH, CASLH

Compare and swap halfword in memory

This instruction 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 comparison is not equal, the architecture permits writing the value read from the location 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.

- If the destination register is not one of `WZR` or `XZR`, `CASAH` and `CASALH` load from memory with acquire semantics.
  - `CASLH` and `CASALH` store to memory with release semantics.
  - `CASH` has neither acquire nor release semantics.

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.

For a `CASH` or `CASAH` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` access to the specified location is likely to occur in the near future. The memory system can respond by taking actions that are expected to enable the subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` access to succeed when it does occur.

A code sequence starting with a `CASH` or `CASAH` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CASH` or `CASAH` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CASH` or `CASAH` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CASH` or `CASAH` either fails its compare or writes the same value back to memory.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSE)

31 30	29 … 23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0 1	0 0 1 0 0 0 1	L	1	Rs	o0	1 1 1 1 1	Rn	Rt
size						Rt2		

Encoding for the CASH variant

Applies when `(L == 0 && o0 == 0)`

```
CASH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASAH variant

Applies when `(L == 1 && o0 == 0)`

```
CASAH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the CASALH variant

Applies when `(L == 1 && o0 == 1)`

```
CASALH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASLH variant

Applies when `(L == 0 && o0 == 1)`

```
CASLH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = L == '1' && t != 31;
constant boolean release = o0 == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          tagchecked, privileged, t, s);
comparevalue = X[s, 16];
newvalue = X[t, 16];

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) data = MemAtomic(address, comparevalue, newvalue, accdesc);
X[s, 32] = ZeroExtend(data, 32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.54 CASP, CASPA, CASPAL, CASPL

Compare and swap pair of words or doublewords in memory

This instruction 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 comparison is not equal, the architecture permits writing the value read from the location 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.
  - `CASP` has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is `<Ws>` and `<W(s+1)>`, or `<Xs>` and `<X(s+1)>`, are restored to the values held in the registers before the instruction was executed.

For a `CASP` or `CASPA` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` access to the specified location is likely to occur in the near future. The memory system can respond by taking actions that are expected to enable the subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` access to succeed when it does occur.

A code sequence starting with a `CASP` or `CASPA` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CASP` or `CASPA` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CASP` or `CASPA` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CASP` or `CASPA` either fails its compare or writes the same value back to memory.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSE)

31	30	29						23	22	21	20				16	15	14				10	9				5	4				0
0	sz	0	0	1	0	0	0	0	L	1	Rs					o0	1	1	1	1	1	Rn					Rt				
																	Rt2														

Encoding for the 32-bit CASP variant

Applies when `(sz == 0 && L == 0 && o0 == 0)`

```
CASP  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 32-bit CASPA variant

Applies when `(sz == 0 && L == 1 && o0 == 0)`

```
CASPA  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASPAL variant

Applies when `(sz == 0 && L == 1 && o0 == 1)`

```
CASPAL  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASPL variant

Applies when `(sz == 0 && L == 0 && o0 == 1)`

```
CASPL  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASP variant

Applies when `(sz == 1 && L == 0 && o0 == 0)`

```
CASP  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPA variant

Applies when `(sz == 1 && L == 1 && o0 == 0)`

```
CASPA  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPAL variant

Applies when `(sz == 1 && L == 1 && o0 == 1)`

```
CASPAL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPL variant

Applies when `(sz == 1 && L == 0 && o0 == 1)`

```
CASPL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
if Rs<0> == '1' || Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sz);
constant boolean acquire = L == '1';
constant boolean release = o0 == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Ws> must be an even-numbered register.

<W(s+1)>

Is the 32-bit name of the second general-purpose register to be compared and loaded.

<Wt>

Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Wt> must be an even-numbered register.

<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, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<X(s+1)>

Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>

Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<X(t+1)>

Is the 64-bit name of the second general-purpose register to be conditionally stored.

Operation

```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

constant bits(datasize) s1 = X[s, datasize];
constant bits(datasize) s2 = X[s+1, datasize];
constant bits(datasize) t1 = X[t, datasize];
constant bits(datasize) t2 = X[t+1, datasize];

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          tagchecked, privileged, t, t+1, s, s+1);

comparevalue = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newvalue     = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if BigEndian(accdesc.acctype) then
    X[s, datasize]   = data<2*datasize-1:datasize>;
    X[s+1, datasize] = data<datasize-1:0>;
else
    X[s, datasize]   = data<datasize-1:0>;
    X[s+1, datasize] = data<2*datasize-1:datasize>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.55 CASPT, CASPAT, CASPALT, CASPLT

Compare and swap pair unprivileged

This instruction reads a pair of 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 comparison is not equal, the architecture permits writing the value read from the location 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.

- `CASPAT` and `CASPALT` load from memory with acquire semantics.
  - `CASPLT` and `CASPALT` store to memory with release semantics.
  - `CASPT` has neither acquire nor release semantics.

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 `<Xs>` and `<X(s+1)>`, are restored to the values held in the registers before the instruction was executed.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For a `CASPT` or `CASPAT` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` access to the specified location is likely to occur in the near future. The memory system can respond by taking actions that are expected to enable the subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` access to succeed when it does occur.

A code sequence starting with a `CASPT` or `CASPAT` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CASPT` or `CASPAT` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CASPT` or `CASPAT` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CASPT` or `CASPAT` either fails its compare or writes the same value back to memory.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the CASPT variant

Applies when `(L == 0 && o0 == 0)`

```
CASPT  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the CASPAT variant

Applies when `(L == 1 && o0 == 0)`

```
CASPAT  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the CASPALT variant

Applies when `(L == 1 && o0 == 1)`

```
CASPALT  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the CASPLT variant

Applies when `(L == 0 && o0 == 1)`

```
CASPLT  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
if Rs<0> == '1' || Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 64;
constant boolean acquire = L == '1';
constant boolean release = o0 == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<X(s+1)>

Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>

Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<X(t+1)>

Is the 64-bit name of the second general-purpose register to be conditionally stored.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

constant bits(datasize) s1 = X[s, datasize];
constant bits(datasize) s2 = X[s+1, datasize];
constant bits(datasize) t1 = X[t, datasize];
constant bits(datasize) t2 = X[t+1, datasize];

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          tagchecked, privileged, t, t+1, s, s+1);

comparevalue = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newvalue     = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

if BigEndian(accdesc.acctype) then
    X[s, datasize]   = data<2*datasize-1:datasize>;
    X[s+1, datasize] = data<datasize-1:0>;
else
    X[s, datasize]   = data<datasize-1:0>;
    X[s+1, datasize] = data<2*datasize-1:datasize>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.56 CAST, CASAT, CASALT, CASLT

Compare and swap unprivileged

This instruction reads a 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 comparison is not equal, the architecture permits writing the value read from the location 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.

- If the destination register is not one of `WZR` or `XZR`, `CASAT` and `CASALT` load from memory with acquire semantics.
  - `CASLT` and `CASALT` store to memory with release semantics.
  - `CAST` has neither acquire nor release semantics.

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 `<Xs>`, is restored to the value held in the register before the instruction was executed.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For a `CAST` or `CASAT` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` access to the specified location is likely to occur in the near future. The memory system can respond by taking actions that are expected to enable the subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` access to succeed when it does occur.

A code sequence starting with a `CAST` or `CASAT` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CAST` or `CASAT` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CAST` or `CASAT` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CAST` or `CASAT` either fails its compare or writes the same value back to memory.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

31	30	29						23	22	21	20			16	15	14				10	9			5	4			0
1	1	0	0	1	0	0	1	1	L	0	Rs				o0	1	1	1	1	1	Rn				Rt			
	sz																	Rt2										

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the CAST variant

Applies when `(L == 0 && o0 == 0)`

```
CAST  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the CASAT variant

Applies when `(L == 1 && o0 == 0)`

```
CASAT  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the CASALT variant

Applies when `(L == 1 && o0 == 1)`

```
CASALT  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the CASLT variant

Applies when `(L == 0 && o0 == 1)`

```
CASLT  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = L == '1' && t != 31;
constant boolean release = o0 == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(64) comparevalue;
bits(64) newvalue;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          tagchecked, privileged, t, s);
comparevalue = X[s, 64];
newvalue = X[t, 64];

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

X[s, 64] = MemAtomic(address, comparevalue, newvalue, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.57 CBB<cc>

Compare bytes and branch

This instruction compares the byte values in two registers, and conditionally branches to a label at a PC-relative offset if the condition is true. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This instruction is used by the pseudo-instructions CBBLE, CBBLO, CBBLS, and CBBLT.

Branch

(FEAT_CMPBR)

Encoding for the Greater than variant

Applies when `(cc == 000)`

```
CBBGT  <Wt>, <Wm>, <label>
```

Encoding for the Greater than or equal variant

Applies when `(cc == 001)`

```
CBBGE  <Wt>, <Wm>, <label>
```

Encoding for the Higher variant

Applies when `(cc == 010)`

```
CBBHI  <Wt>, <Wm>, <label>
```

Encoding for the Higher or same variant

Applies when `(cc == 011)`

```
CBBHS  <Wt>, <Wm>, <label>
```

Encoding for the Equal variant

Applies when `(cc == 110)`

```
CBBEQ  <Wt>, <Wm>, <label>
```

Encoding for the Not equal variant

Applies when `(cc == 111)`

```
CBBNE  <Wt>, <Wm>, <label>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CMPBR) then EndOfDecode(Decode_UNDEF);
constant integer datasize = 8 << UInt(H);
constant integer t = UInt(Rt);
constant integer m = UInt(Rm);
constant bits(64) offset = SignExtend(imm9:'00', 64);
CmpOp op;
boolean unsigned;

case cc of
    when '000' op = Cmp_GT; unsigned = FALSE;
    when '001' op = Cmp_GE; unsigned = FALSE;
    when '010' op = Cmp_GT; unsigned = TRUE;
    when '011' op = Cmp_GE; unsigned = TRUE;
    when '110' op = Cmp_EQ; unsigned = TRUE;
    when '111' op = Cmp_NE; unsigned = TRUE;
    otherwise EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

```
constant bits(datasize) operand1 = X[t, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant boolean branch_conditional = TRUE;

constant integer value1 = if unsigned then UInt(operand1) else SInt(operand1);
constant integer value2 = if unsigned then UInt(operand2) else SInt(operand2);
boolean cond;
case op of
    when Cmp_EQ cond = value1 == value2;
    when Cmp_NE cond = value1 != value2;
    when Cmp_GE cond = value1 >= value2;
    when Cmp_GT cond = value1 >  value2;

if cond then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.58 CBBLE

Compare signed less than or equal to bytes and branch

This instruction compares the signed byte values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than or equal to the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBB<cc>. This means:

- The encodings in this description are named to match the encodings of CBB<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBB<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 0 1	Rm	1	0	imm9	Rt
			cc			H		

Encoding

```
CBBLE  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBBGE  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBB<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.59 CBBLO

Compare unsigned lower than bytes and branch

This instruction compares the unsigned byte values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBB<cc>. This means:

- The encodings in this description are named to match the encodings of CBB<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBB<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 1 0 (cc)	Rm	1	0 (H)	imm9	Rt

Encoding

```
CBBLO  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBBHI  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBB<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.60 CBBLS

Compare unsigned lower than or same as bytes and branch

This instruction compares the unsigned byte values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than or the same as the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBB<cc>. This means:

- The encodings in this description are named to match the encodings of CBB<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBB<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 1 1 (cc)	Rm	1	0 (H)	imm9	Rt

Encoding

```
CBBLS  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBBHS  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBB<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.61 CBBLT

Compare signed less than bytes and branch

This instruction compares the signed byte values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBB<cc>. This means:

- The encodings in this description are named to match the encodings of CBB<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBB<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	29	28	26	25	24	23	21	20	16	15	14	13	5	4	0
0 1 1		1 0 1		0 0		0 0 0		Rm		1	0	imm9		Rt	
						cc					H				

Encoding

```
CBBLT  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBBGT  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBB<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.62 CB<cc> (immediate)

Compare register with immediate and branch

This instruction compares the value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the comparison is true. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This instruction is used by the pseudo-instructions CBGE (immediate), CBHS (immediate), CBLE (immediate), and CBLS (immediate).

Branch

(FEAT_CMPBR)

31	30	24	23 21	20 15	14	13 5	4 0
sf	1 1 1 0 1 0 1		cc	imm6	0	imm9	Rt

Encoding for the 32-bit greater than variant

Applies when `(sf == 0 && cc == 000)`

```
CBGT  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit less than variant

Applies when `(sf == 0 && cc == 001)`

```
CBLT  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit higher variant

Applies when `(sf == 0 && cc == 010)`

```
CBHI  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit lower variant

Applies when `(sf == 0 && cc == 011)`

```
CBLO  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit equal variant

Applies when `(sf == 0 && cc == 110)`

```
CBEQ  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit not equal variant

Applies when `(sf == 0 && cc == 111)`

```
CBNE  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit greater than variant

Applies when `(sf == 1 && cc == 000)`

```
CBGT  <Xt>, #<imm>, <label>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit less than variant

Applies when `(sf == 1 && cc == 001)`

```
CBLT  <Xt>, #<imm>, <label>
```

Encoding for the 64-bit higher variant

Applies when `(sf == 1 && cc == 010)`

```
CBHI  <Xt>, #<imm>, <label>
```

Encoding for the 64-bit lower variant

Applies when `(sf == 1 && cc == 011)`

```
CBLO  <Xt>, #<imm>, <label>
```

Encoding for the 64-bit equal variant

Applies when `(sf == 1 && cc == 110)`

```
CBEQ  <Xt>, #<imm>, <label>
```

Encoding for the 64-bit not equal variant

Applies when `(sf == 1 && cc == 111)`

```
CBNE  <Xt>, #<imm>, <label>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CMPBR) then EndOfDecode(Decode_UNDEF);
constant integer datasize = 32 << UInt(sf);
constant integer t = UInt(Rt);
constant bits(64) offset = SignExtend(imm9:'00', 64);
CmpOp op;
boolean unsigned;

case cc of
    when '000' op = Cmp_GT; unsigned = FALSE;
    when '001' op = Cmp_LT; unsigned = FALSE;
    when '010' op = Cmp_GT; unsigned = TRUE;
    when '011' op = Cmp_LT; unsigned = TRUE;
    when '110' op = Cmp_EQ; unsigned = TRUE;
    when '111' op = Cmp_NE; unsigned = TRUE;
    otherwise EndOfDecode(Decode_UNDEF);
constant integer value2 = UInt(imm6);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<imm>

Is an unsigned immediate, in the range 0 to 63, encoded in the “imm6” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[t, datasize];
constant boolean branch_conditional = TRUE;

constant integer value1 = if unsigned then UInt(operand1) else SInt(operand1);
boolean cond;
case op of
    when Cmp_EQ cond = value1 == value2;
    when Cmp_NE cond = value1 != value2;
    when Cmp_LT cond = value1 <  value2;
    when Cmp_GT cond = value1 >  value2;

if cond then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.63 CB<cc> (register)

Compare registers and branch

This instruction compares the values in two registers, and conditionally branches to a label at a PC-relative offset if the condition is true. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This instruction is used by the pseudo-instructions CBLE (register), CBLO (register), CBLS (register), and CBLT (register).

Branch

(FEAT_CMPBR)

31	30	24	23 21	20 16	15 14	13 5	4 0
sf	1 1 1 0 1 0 0		cc	Rm	0 0	imm9	Rt

Encoding for the 32-bit greater than variant

Applies when `(sf == 0 && cc == 000)`

```
CBGT  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit greater than or equal variant

Applies when `(sf == 0 && cc == 001)`

```
CBGE  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit higher variant

Applies when `(sf == 0 && cc == 010)`

```
CBHI  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit higher or same variant

Applies when `(sf == 0 && cc == 011)`

```
CBHS  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit equal variant

Applies when `(sf == 0 && cc == 110)`

```
CBEQ  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit not equal variant

Applies when `(sf == 0 && cc == 111)`

```
CBNE  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit greater than variant

Applies when `(sf == 1 && cc == 000)`

```
CBGT  <Xt>, <Xm>, <label>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit greater than or equal variant

Applies when `(sf == 1 && cc == 001)`

```
CBGE  <Xt>, <Xm>, <label>
```

Encoding for the 64-bit higher variant

Applies when `(sf == 1 && cc == 010)`

```
CBHI  <Xt>, <Xm>, <label>
```

Encoding for the 64-bit higher or same variant

Applies when `(sf == 1 && cc == 011)`

```
CBHS  <Xt>, <Xm>, <label>
```

Encoding for the 64-bit equal variant

Applies when `(sf == 1 && cc == 110)`

```
CBEQ  <Xt>, <Xm>, <label>
```

Encoding for the 64-bit not equal variant

Applies when `(sf == 1 && cc == 111)`

```
CBNE  <Xt>, <Xm>, <label>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CMPBR) then EndOfDecode(Decode_UNDEF);
constant integer datasize = 32 << UInt(sf);
constant integer t = UInt(Rt);
constant integer m = UInt(Rm);
constant bits(64) offset = SignExtend(imm9:'00', 64);
CmpOp op;
boolean unsigned;

case cc of
    when '000' op = Cmp_GT; unsigned = FALSE;
    when '001' op = Cmp_GE; unsigned = FALSE;
    when '010' op = Cmp_GT; unsigned = TRUE;
    when '011' op = Cmp_GE; unsigned = TRUE;
    when '110' op = Cmp_EQ; unsigned = TRUE;
    when '111' op = Cmp_NE; unsigned = TRUE;
    otherwise EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand1 = X[t, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant boolean branch_conditional = TRUE;

constant integer value1 = if unsigned then UInt(operand1) else SInt(operand1);
constant integer value2 = if unsigned then UInt(operand2) else SInt(operand2);
boolean cond;
case op of
    when Cmp_EQ cond = value1 == value2;
    when Cmp_NE cond = value1 != value2;
    when Cmp_GE cond = value1 >= value2;
    when Cmp_GT cond = value1 >  value2;

if cond then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.64 CBGE (immediate)

Compare signed greater than or equal to immediate and branch

This instruction compares the signed value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the register value is greater than or equal to the immediate. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (immediate). This means:

- The encodings in this description are named to match the encodings of CB<cc> (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20	15	14	13	5	4	0
sf	1	1	1	0	1	0	1	0	0	0	imm6		0	imm9		Rt	
										cc							

Encoding for the 32-bit greater than variant

Applies when `(sf == 0)`

```
CBGE  <Wt>, #<immp1>, <label>
```

is equivalent to

```
CBGT  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit greater than variant

Applies when `(sf == 1)`

```
CBGE  <Xt>, #<immp1>, <label>
```

is equivalent to

```
CBGT  <Xt>, #<imm>, <label>
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<immp1>

Is an unsigned immediate, in the range 1 to 64, encoded as “imm6” plus 1.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of CB<cc> (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.65 CBH<cc>

Compare halfwords and branch

This instruction compares the halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the condition is true. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This instruction is used by the pseudo-instructions CBHLE, CBHLO, CBHLS, and CBHLT.

Branch

(FEAT_CMPBR)

31	24	23	21	20	16	15	14	13	5	4	0
0 1 1 1 0 1 0 0		cc		Rm		1	1	imm9		Rt	
							H				

Encoding for the Greater than variant

Applies when `(cc == 000)`

```
CBHGT  <Wt>, <Wm>, <label>
```

Encoding for the Greater than or equal variant

Applies when `(cc == 001)`

```
CBHGE  <Wt>, <Wm>, <label>
```

Encoding for the Higher variant

Applies when `(cc == 010)`

```
CBHHI  <Wt>, <Wm>, <label>
```

Encoding for the Higher or same variant

Applies when `(cc == 011)`

```
CBHHS  <Wt>, <Wm>, <label>
```

Encoding for the Equal variant

Applies when `(cc == 110)`

```
CBHEQ  <Wt>, <Wm>, <label>
```

Encoding for the Not equal variant

Applies when `(cc == 111)`

```
CBHNE  <Wt>, <Wm>, <label>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CMPBR) then EndOfDecode(Decode_UNDEF);
constant integer datasize = 8 << UInt(H);
constant integer t = UInt(Rt);
constant integer m = UInt(Rm);
constant bits(64) offset = SignExtend(imm9:'00', 64);
CmpOp op;
boolean unsigned;

case cc of
    when '000' op = Cmp_GT; unsigned = FALSE;
    when '001' op = Cmp_GE; unsigned = FALSE;
    when '010' op = Cmp_GT; unsigned = TRUE;
    when '011' op = Cmp_GE; unsigned = TRUE;
    when '110' op = Cmp_EQ; unsigned = TRUE;
    when '111' op = Cmp_NE; unsigned = TRUE;
    otherwise EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

```
constant bits(datasize) operand1 = X[t, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant boolean branch_conditional = TRUE;

constant integer value1 = if unsigned then UInt(operand1) else SInt(operand1);
constant integer value2 = if unsigned then UInt(operand2) else SInt(operand2);
boolean cond;
case op of
    when Cmp_EQ cond = value1 == value2;
    when Cmp_NE cond = value1 != value2;
    when Cmp_GE cond = value1 >= value2;
    when Cmp_GT cond = value1 >  value2;

if cond then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.66 CBHLE

Compare signed less than or equal to halfwords and branch

This instruction compares the signed halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than or equal to the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBH<cc>. This means:

- The encodings in this description are named to match the encodings of CBH<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBH<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 0 1 (cc)	Rm	1	1 (H)	imm9	Rt

Encoding

```
CBHLE  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHGE  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBH<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.67 CBHLO

Compare unsigned lower than halfwords and branch

This instruction compares the unsigned halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBH<cc>. This means:

- The encodings in this description are named to match the encodings of CBH<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBH<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 1 0 (cc)	Rm	1	1 (H)	imm9	Rt

Encoding

```
CBHLO  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHHI  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBH<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.68 CBHLS

Compare unsigned lower than or same as halfwords and branch

This instruction compares the unsigned halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than or the same as the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBH<cc>. This means:

- The encodings in this description are named to match the encodings of CBH<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBH<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 1 1 (cc)	Rm	1	1 (H)	imm9	Rt

Encoding

```
CBHLS  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHHS  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBH<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.69 CBHLT

Compare signed less than halfwords and branch

This instruction compares the signed halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBH<cc>. This means:

- The encodings in this description are named to match the encodings of CBH<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBH<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 0 0 (cc)	Rm	1	1 (H)	imm9	Rt

Encoding

```
CBHLT  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHGT  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBH<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.70 CBHS (immediate)

Compare unsigned higher than or same as immediate and branch

This instruction compares the unsigned value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the register value is higher than or the same as the immediate. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (immediate). This means:

- The encodings in this description are named to match the encodings of CB<cc> (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20		15	14	13		5	4		0
sf	1	1	1	0	1	0	1	0	1	0	imm6			0	imm9			Rt		
										cc										

Encoding for the 32-bit higher variant

Applies when `(sf == 0)`

```
CBHS  <Wt>, #<immp1>, <label>
```

is equivalent to

```
CBHI  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit higher variant

Applies when `(sf == 1)`

```
CBHS  <Xt>, #<immp1>, <label>
```

is equivalent to

```
CBHI  <Xt>, #<imm>, <label>
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<immp1>

Is an unsigned immediate, in the range 1 to 64, encoded as “imm6” plus 1.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of CB<cc> (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.71 CBLE (immediate)

Compare signed less than or equal to immediate and branch

This instruction compares the signed value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the register value is less than or equal to the immediate. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (immediate). This means:

- The encodings in this description are named to match the encodings of CB<cc> (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20	15	14	13	5	4	0
sf	1	1	1	0	1	0	1	0	0	1	imm6		0	imm9		Rt	
										cc							

Encoding for the 32-bit less than variant

Applies when `(sf == 0)`

```
CBLE  <Wt>, #<imms1>, <label>
```

is equivalent to

```
CBLT  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit less than variant

Applies when `(sf == 1)`

```
CBLE  <Xt>, #<imms1>, <label>
```

is equivalent to

```
CBLT  <Xt>, #<imm>, <label>
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<imms1>

Is a signed immediate, in the range -1 to 62, encoded as “imm6” minus 1.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of CB<cc> (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.72 CBLE (register)

Compare signed less than or equal to register and branch

This instruction compares the signed values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than or equal to the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (register). This means:

- The encodings in this description are named to match the encodings of CB<cc> (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20			16	15	14	13					5	4				0
sf	1	1	1	0	1	0	0	0	0	1	Rm				0	0	imm9						Rt				
										cc																	

Encoding for the 32-bit greater than or equal variant

Applies when `(sf == 0)`

```
CBLE  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBGE  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit greater than or equal variant

Applies when `(sf == 1)`

```
CBLE  <Xm>, <Xt>, <label>
```

is equivalent to

```
CBGE  <Xt>, <Xm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

The description of CB<cc> (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.73 CBLO (register)

Compare unsigned lower than register and branch

This instruction compares the unsigned values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (register). This means:

- The encodings in this description are named to match the encodings of CB<cc> (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20			16	15	14	13							5	4							0
sf	1	1	1	0	1	0	0	0	1	0	Rm				0	0	imm9								Rt						
										cc																					

Encoding for the 32-bit higher variant

Applies when `(sf == 0)`

```
CBLO  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHI  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit higher variant

Applies when `(sf == 1)`

```
CBLO  <Xm>, <Xt>, <label>
```

is equivalent to

```
CBHI  <Xt>, <Xm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

The description of CB<cc> (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.74 CBLS (immediate)

Compare unsigned lower than or same as immediate and branch

This instruction compares the unsigned value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the register value is lower than or the same as the immediate. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (immediate). This means:

- The encodings in this description are named to match the encodings of CB<cc> (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20		15	14	13		5	4		0
sf	1	1	1	0	1	0	1	0	1	1	imm6			0	imm9			Rt		
										cc										

Encoding for the 32-bit lower variant

Applies when `(sf == 0)`

```
CBLS  <Wt>, #<imms1>, <label>
```

is equivalent to

```
CBLO  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit lower variant

Applies when `(sf == 1)`

```
CBLS  <Xt>, #<imms1>, <label>
```

is equivalent to

```
CBLO  <Xt>, #<imm>, <label>
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<imms1>

Is a signed immediate, in the range -1 to 62, encoded as “imm6” minus 1.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of CB<cc> (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.75 CBLS (register)

Compare unsigned lower than or same as register and branch

This instruction compares the unsigned values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than or the same as the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (register). This means:

- The encodings in this description are named to match the encodings of CB<cc> (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20			16	15	14	13					5	4				0
sf	1	1	1	0	1	0	0	0	1	1	Rm				0	0	imm9						Rt				
										cc																	

Encoding for the 32-bit higher or same variant

Applies when `(sf == 0)`

```
CBLS  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHS  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit higher or same variant

Applies when `(sf == 1)`

```
CBLS  <Xm>, <Xt>, <label>
```

is equivalent to

```
CBHS  <Xt>, <Xm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

The description of CB<cc> (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.76 CBLT (register)

Compare signed less than register and branch

This instruction compares the signed values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (register). This means:

- The encodings in this description are named to match the encodings of CB<cc> (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20		16	15	14	13		5	4		0
sf	1	1	1	0	1	0	0	0	0	0	Rm			0	0	imm9			Rt		
										cc											

Encoding for the 32-bit greater than variant

Applies when `(sf == 0)`

```
CBLT  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBGT  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit greater than variant

Applies when `(sf == 1)`

```
CBLT  <Xm>, <Xt>, <label>
```

is equivalent to

```
CBGT  <Xt>, <Xm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

The description of CB<cc> (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.77 CBNZ

Compare and branch on nonzero

This instruction 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. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CBNZ  <Wt>, <label>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CBNZ  <Xt>, <label>
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer datasize = 32 << UInt(sf);
constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

```
constant boolean branch_conditional = TRUE;
constant bits(datasize) operand1 = X[t, datasize];
if !IsZero(operand1) then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.78 CBZ

Compare and branch on zero

This instruction compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CBZ  <Wt>, <label>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CBZ  <Xt>, <label>
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer datasize = 32 << UInt(sf);
constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

```
constant boolean branch_conditional = TRUE;
constant bits(datasize) operand1 = X[t, datasize];
if IsZero(operand1) then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.79 CCMN (immediate)

Conditional compare negative (immediate)

This instruction 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.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CCMN  <Wn>, #<imm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CCMN  <Xn>, #<imm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
bits(4) flags = nzcv;
constant bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler Symbols

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is a five bit unsigned (positive) immediate encoded in the “imm5” field.

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionHolds(condition) then
    constant bits(datasize) operand1 = X[n, datasize];
    constant bits(datasize) operand2 = imm;
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.80 CCMN (register)

Conditional compare negative (register)

This instruction 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.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CCMN  <Wn>, <Wm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CCMN  <Xn>, <Xm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionHolds(condition) then
    constant bits(datasize) operand1 = X[n, datasize];
    constant bits(datasize) operand2 = X[m, datasize];
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.81 CCMP (immediate)

Conditional compare (immediate)

This instruction 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.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CCMP  <Wn>, #<imm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CCMP  <Xn>, #<imm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
bits(4) flags = nzcv;
constant bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler Symbols

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is a five bit unsigned (positive) immediate encoded in the “imm5” field.

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionHolds(condition) then
    constant bits(datasize) operand1 = X[n, datasize];
    constant bits(datasize) operand2 = imm;
    (-, flags) = AddWithCarry(operand1, NOT(operand2), '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.82 CCMP (register)

Conditional compare (register)

This instruction 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.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CCMP  <Wn>, <Wm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CCMP  <Xn>, <Xm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionHolds(condition) then
    constant bits(datasize) operand1 = X[n, datasize];
    constant bits(datasize) operand2 = X[m, datasize];
    (-, flags) = AddWithCarry(operand1, NOT(operand2), '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.83 CFINV

Invert carry flag

This instruction inverts the value of the PSTATE.C flag.

System

(FEAT_FlagM)

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	0 0 0	0 1 0 0	(0) (0) (0) (0)	0 0 0	1 1 1 1 1
			op1		CRm	op2	Rt

Encoding

```
CFINV
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF);
```

Operation

```
PSTATE.C = NOT(PSTATE.C);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.84 CFP

Control flow prediction restriction by context

This instruction prevents control flow predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see CFP RCTX, Control Flow Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 1	1 0 0	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
CFP  RCTX, <Xt>
```

is equivalent to

```
SYS  #3, C7, C3, #4, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.85 CHKFEAT

Check feature status

This instruction indicates the status of features. For more information, see Check Feature.

If FEAT_CHK is not implemented, this instruction executes as a `NOP`.

System

(FEAT_CHK)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 1 0 1 (CRm)	0 0 0 (op2)	1 1 1 1 1

Encoding

```
CHKFEAT  X16
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CHK) then EndOfDecode(Decode_NOP);
```

Operation

```
X[16, 64] = AArch64.ChkFeat(X[16, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.86 CINC

Conditional increment

This instruction 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.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CINC  <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSINC  <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CINC  <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSINC  <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational Information

The description of CSINC gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.87 CINV

Conditional invert

This instruction 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.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CINV  <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSINV  <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CINV  <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSINV  <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational Information

The description of CSINV gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.88 CLRBHB

Clear branch history

This instruction can be used to clear the branch history.

For more information, see Branch history.

System

(FEAT_CLRBHB)

31–29	28–26	25–12	11–8	7–5	4–0
110	101	0100000011001 0	0010 (CRm)	110 (op2)	11111

Encoding

```
CLRBHB
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CLRBHB) then EndOfDecode(Decode_NOP);
```

Operation

```
Hint_CLRBHB();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.89 CLREX

Clear exclusive

This instruction clears the local monitor of the executing PE.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	0 1 0 (op2)	1 1 1 1 1 (Rt)

Encoding

```
CLREX  {#<imm>}
```

Decode for this encoding

```
// CRm field is ignored
```

Assembler Symbols

<imm>

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the “CRm” field.

Operation

```
ClearExclusiveLocal(ProcessorID());
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.90 CLS

Count leading sign bits

This instruction counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.

31	30	29	28	21	20	16	15	11	10	9	5	4	0
sf	1	0	1 1 0 1 0 1 1 0		0 0 0 0 0		0 0 0 1 0		1	Rn		Rd	
		S				opcode2			op				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CLS  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CLS  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = CountLeadingSignBits(operand1);

X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.91 CLZ

Count leading zeros

This instruction counts the number of consecutive binary zero bits, starting from the most significant bit in the source register, and places the count in the destination register.

31	30	29	28							21	20					16	15					11	10	9				5	4				0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0		0	0	0	1	0		0	Rn					Rd				
		S														opcode2							op										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CLZ  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CLZ  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = CountLeadingZeroBits(operand1);

X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.92 CMN (extended register)

Compare negative (extended register)

This instruction 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.

This is an alias of ADDS (extended register). This means:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28				24	23	22	21	20		16	15		13	12		10	9		5	4				0
sf	0	1	0	1	0	1	1	0	0	1	Rm			option			imm3			Rn			1	1	1	1	1
op		S						opt																			Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

is equivalent to

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

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

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

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

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

<amount>

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

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<R>
x11	X
10x	W
110	W

<m>

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

Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (extended register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.93 CMN (immediate)

Compare negative (immediate)

This instruction adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (immediate). This means:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

is equivalent to

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

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sh	<shift>
0	LSL #0
1	LSL #12

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (immediate) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.94 CMN (shifted register)

Compare negative (shifted register)

This instruction adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28				24	23	22	21	20			16	15					10	9				5	4				0
sf	0	1	0	1	0	1	1	shift		0	Rm				imm6						Rn					1	1	1	1	1
op		S																												Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CMN  <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

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

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CMN  <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (shifted register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.95 CMP (extended register)

Compare (extended register)

This instruction 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.

This is an alias of SUBS (extended register). This means:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	24	23	22	21	20	16	15	13	12	10	9	5	4	0
sf	1	1	0 1 0 1 1		0	0	1	Rm		option		imm3		Rn		1 1 1 1 1	
op		S			opt											Rd	

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

is equivalent to

```
SUBS  WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

is equivalent to

```
SUBS  XZR, <Xn|SP>, <R><m>{, <extend> {#amount>}}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

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

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

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

<amount>

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

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<R>
x11	X
10x	W
110	W

<m>

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

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (extended register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.96 CMP (immediate)

Compare (immediate)

This instruction subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (immediate). This means:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

is equivalent to

```
SUBS  WZR, <Wn|WSP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

is equivalent to

```
SUBS  XZR, <Xn|SP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sh	<shift>
0	LSL #0
1	LSL #12

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (immediate) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.97 CMP (shifted register)

Compare (shifted register)

This instruction subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	24	23 22	21	20	16	15	10	9	5	4	0
sf	1	1	0 1 0 1 1		shift	0	Rm		imm6		Rn		1 1 1 1 1	
	op	S												Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CMP  <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS  WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CMP  <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS  XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (shifted register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.98 CMPP

Compare with tag

This instruction 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.

This is an alias of SUBPS. This means:

- The encodings in this description are named to match the encodings of SUBPS.
- The description of SUBPS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_MTE)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	0	1	1	1 0 1	0 1 1 0	Rm	0 0 0 0 0 0	Rn	1 1 1 1 1
sf		S					opcode		Rd

Encoding

```
CMPP  <Xn|SP>, <Xm|SP>
```

is equivalent to

```
SUBPS  <Xd>, <Xn|SP>, <Xm|SP>
```

and is always the preferred disassembly.

Assembler Symbols

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Rm” field.

Operation

The description of SUBPS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.99 CNEG

Conditional negate

This instruction 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.

This is an alias of CSNEG. This means:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	1	0	1	1	0	1	0	1	0	0	Rm					!=111x					0	1	Rn					Rd				
op		S														cond						o2										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CNEG  <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSNEG  <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CNEG  <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSNEG  <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSNEG gives the operational pseudocode for this instruction.

Operational Information

The description of CSNEG gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.100 CNT

Count bits

This instruction counts the number of binary one bits in the value of the source register, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28							21	20					16	15					10	9			5	4			0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0		0	0	0	1	1	1	Rn				Rd			
		S									opcode2									opcode										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CNT  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CNT  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer datasize = 32 << UInt(sf);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = BitCount(operand1);
X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.101 COSP

Clear other speculative prediction restriction by context

This instruction prevents predictions, other than Cache prefetch, Control flow, and Data Value predictions, that predict execution addresses based on information gathered from earlier execution within a particular execution context. Predictions, other than Cache prefetch, Control flow, and Data Value predictions, determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control any speculative access occurring after the instruction is complete and synchronized.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES2)

Encoding

```
COSP  RCTX, <Xt>
```

is equivalent to

```
SYS  #3, C7, C3, #6, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.102 CPP

Cache prefetch prediction restriction by context

This instruction prevents cache allocation predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

For more information, see CPP RCTX, Cache Prefetch Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 1	1 1 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
CPP  RCTX, <Xt>
```

is equivalent to

```
SYS  #3, C7, C3, #7, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.103 CPYFP, CPYFM, CPYFE

Memory copy forward-only

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFP, then CPYFM, and then CPYFE.

CPYFP performs some preconditioning of the arguments suitable for using the CPYFM instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFM copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFE copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFP:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFP, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFP, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFM, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFM, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFE, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFE, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 0 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFP  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFM  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFE  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

<Xs>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.104 CPYFPN, CPYFMN, CPYFEN

Memory copy forward-only, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPN, then CPYFMN, and then CPYFEN.

CPYFPN performs some preconditioning of the arguments suitable for using the CPYFMN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 1 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

<Xs>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    memcpy.forward = TRUE;
    boolean fault  = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= -1 * memcpy.stagecpysize;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                    memcpy.fromaddress + memcpy.cpysize,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress  = memcpy.fromaddress + B;
                memcpy.toaddress    = memcpy.toaddress   + B;
                memcpy.cpysize      = memcpy.cpysize     - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

    UpdateCpyRegisters(memcpy, fault, copied);

    if fault then
        if IsFault(memaddrdesc) then
            AArch64.Abort(memaddrdesc.fault);

        if IsFault(memstatus) then
            constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
            HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

    if memcpy.stage == MOPSStage_Prologue then
        PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.105 CPYFPRN, CPYFMRN, CPYFERN

Memory copy forward-only, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRN, then CPYFMRN, and then CPYFERN.

CPYFPRN performs some preconditioning of the arguments suitable for using the CPYFMRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 0 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

<Xs>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.106 CPYFPRT, CPYFMRT, CPYFERT

Memory copy forward-only, reads unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRT, then CPYFMRT, and then CPYFERT.

CPYFPRT performs some preconditioning of the arguments suitable for using the CPYFMRT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERT copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRT:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRT, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRT, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERT  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.107 CPYFPRTN, CPYFMRTN, CPYFERTN

Memory copy forward-only, reads unprivileged, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTN, then CPYFMRTN, and then CPYFERTN.

CPYFPRTN performs some preconditioning of the arguments suitable for using the CPYFMRTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERTN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRTN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRTN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRTN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 1 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERTN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.108 CPYFPRTRN, CPYFMRTRN, CPYFERTRN

Memory copy forward-only, reads unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTRN, then CPYFMRTRN, and then CPYFERTRN.

CPYFPRTRN performs some preconditioning of the arguments suitable for using the CPYFMRTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERTRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRTRN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRTRN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRTRN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 0 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.109 CPYFPRTWN, CPYFMRTWN, CPYFERTWN

Memory copy forward-only, reads unprivileged, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTWN, then CPYFMRTWN, and then CPYFERTWN.

CPYFPRTWN performs some preconditioning of the arguments suitable for using the CPYFMRTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERTWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRTWN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRTWN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRTWN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 1 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.110 CPYFPT, CPYFMT, CPYFET

Memory copy forward-only, reads and writes unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPT, then CPYFMT, and then CPYFET.

CPYFPT performs some preconditioning of the arguments suitable for using the CPYFMT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFET copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPT:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPT, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPT, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFET, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFET, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 0 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFET  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.111 CPYFPTN, CPYFMTN, CPYFETN

Memory copy forward-only, reads and writes unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTN, then CPYFMTN, and then CPYFETN.

CPYFPTN performs some preconditioning of the arguments suitable for using the CPYFMTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFETN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPTN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPTN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPTN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

For CPYFETN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFETN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFETN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.112 CPYFPTRN, CPYFMTRN, CPYFETRN

Memory copy forward-only, reads and writes unprivileged, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTRN, then CPYFMTRN, and then CPYFETRN.

CPYFPTRN performs some preconditioning of the arguments suitable for using the CPYFMTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFETRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPTRN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPTRN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPTRN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFETRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFETRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 0 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFETRN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.113 CPYFPTWN, CPYFMTWN, CPYFETWN

Memory copy forward-only, reads and writes unprivileged, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTWN, then CPYFMTWN, and then CPYFETWN.

CPYFPTWN performs some preconditioning of the arguments suitable for using the CPYFMTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFETWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPTWN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPTWN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPTWN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFETWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFETWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 1 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFETWN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.114 CPYFPWN, CPYFMWN, CPYFEWN

Memory copy forward-only, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWN, then CPYFMWN, and then CPYFEWN.

CPYFPWN performs some preconditioning of the arguments suitable for using the CPYFMWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 1 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

<Xs>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    memcpy.forward = TRUE;
    boolean fault  = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= -1 * memcpy.stagecpysize;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                    memcpy.fromaddress + memcpy.cpysize,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress  = memcpy.fromaddress + B;
                memcpy.toaddress    = memcpy.toaddress   + B;
                memcpy.cpysize      = memcpy.cpysize     - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

    UpdateCpyRegisters(memcpy, fault, copied);

    if fault then
        if IsFault(memaddrdesc) then
            AArch64.Abort(memaddrdesc.fault);

        if IsFault(memstatus) then
            constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
            HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

    if memcpy.stage == MOPSStage_Prologue then
        PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.115 CPYFPWT, CPYFMWT, CPYFEWT

Memory copy forward-only, writes unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWT, then CPYFMWT, and then CPYFEWT.

CPYFPWT performs some preconditioning of the arguments suitable for using the CPYFMWT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWT copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWT:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWT, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWT, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 0 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWT  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.116 CPYFPWTN, CPYFMWTN, CPYFEWTN

Memory copy forward-only, writes unprivileged, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTN, then CPYFMWTN, and then CPYFEWTN.

CPYFPWTN performs some preconditioning of the arguments suitable for using the CPYFMWTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWTN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWTN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWTN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWTN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 1 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.117 CPYFPWTRN, CPYFMWTRN, CPYFEWTRN

Memory copy forward-only, writes unprivileged, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTRN, then CPYFMWTRN, and then CPYFEWTRN.

CPYFPWTRN performs some preconditioning of the arguments suitable for using the CPYFMWTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWTRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWTRN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWTRN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWTRN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 0 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.118 CPYFPWTWN, CPYFMWTWN, CPYFEWTWN

Memory copy forward-only, writes unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTWN, then CPYFMWTWN, and then CPYFEWTWN.

CPYFPWTWN performs some preconditioning of the arguments suitable for using the CPYFMWTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWTWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWTWN:

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWTWN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWTWN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 1 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize     = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
memcpy.forward = TRUE;
boolean fault  = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.119 CPYP, CPYM, CPYE

Memory copy

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYP, then CPYM, and then CPYE.

CPYP performs some preconditioning of the arguments suitable for using the CPYM instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYM copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYE copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYP:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYP, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYP, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYM, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYM, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYE, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYE, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 0 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYP  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYM  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYE  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
boolean fault = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.120 CPYPN, CPYMN, CPYEN

Memory copy, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPN, then CPYMN, and then CPYEN.

CPYPN performs some preconditioning of the arguments suitable for using the CPYMN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYEN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 1 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
boolean fault = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.121 CPYPRN, CPYMRN, CPYERN

Memory copy, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRN, then CPYMRN, and then CPYERN.

CPYPRN performs some preconditioning of the arguments suitable for using the CPYMRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYERN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 0 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
boolean fault = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.122 CPYPRT, CPYMRT, CPYERT

Memory copy, reads unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRT, then CPYMRT, and then CPYERT.

CPYPRT performs some preconditioning of the arguments suitable for using the CPYMRT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERT copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRT:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRT, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRT, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYERT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 0 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.123 CPYPRTN, CPYMRTN, CPYERTN

Memory copy, reads unprivileged, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTN, then CPYMRTN, and then CPYERTN.

CPYPRTN performs some preconditioning of the arguments suitable for using the CPYMRTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERTN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRTN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRTN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRTN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYERTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 1 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.124 CPYPRTRN, CPYMRTRN, CPYERTRN

Memory copy, reads unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTRN, then CPYMRTRN, and then CPYERTRN.

CPYPRTRN performs some preconditioning of the arguments suitable for using the CPYMRTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERTRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRTRN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRTRN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRTRN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYERTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 0 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.125 CPYPRTWN, CPYMRTWN, CPYERTWN

Memory copy, reads unprivileged, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTWN, then CPYMRTWN, and then CPYERTWN.

CPYPRTWN performs some preconditioning of the arguments suitable for using the CPYMRTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERTWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRTWN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRTWN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRTWN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYERTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 1 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.126 CPYPT, CPYMT, CPYET

Memory copy, reads and writes unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPT, then CPYMT, and then CPYET.

CPYPT performs some preconditioning of the arguments suitable for using the CPYMT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYET copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPT:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPT, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPT, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYET, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYET, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 0 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYET  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.127 CPYPTN, CPYMTN, CPYETN

Memory copy, reads and writes unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTN, then CPYMTN, and then CPYETN.

CPYPTN performs some preconditioning of the arguments suitable for using the CPYMTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYETN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPTN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPTN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPTN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYETN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYETN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 1 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYETN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.128 CPYPTRN, CPYMTRN, CPYETRN

Memory copy, reads and writes unprivileged, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTRN, then CPYMTRN, and then CPYETRN.

CPYPTRN performs some preconditioning of the arguments suitable for using the CPYMTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYETRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPTRN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPTRN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPTRN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYETRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYETRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 0 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYETRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.129 CPYPTWN, CPYMTWN, CPYETWN

Memory copy, reads and writes unprivileged, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTWN, then CPYMTWN, and then CPYETWN.

CPYPTWN performs some preconditioning of the arguments suitable for using the CPYMTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYETWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPTWN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPTWN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPTWN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYETWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYETWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 1 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYETWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.130 CPYPWN, CPYMWN, CPYEWN

Memory copy, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWN, then CPYMWN, and then CPYEWN.

CPYPWN performs some preconditioning of the arguments suitable for using the CPYMWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYEWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 1 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                 else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
boolean fault = FALSE;
MOPSBlockSize B = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.131 CPYPWT, CPYMWT, CPYEWT

Memory copy, writes unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWT, then CPYMWT, and then CPYEWT.

CPYPWT performs some preconditioning of the arguments suitable for using the CPYMWT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWT copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWT:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWT, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWT, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYEWT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 0 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.132 CPYPWTN, CPYMWTN, CPYEWTN

Memory copy, writes unprivileged, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTN, then CPYMWTN, and then CPYEWTN.

CPYPWTN performs some preconditioning of the arguments suitable for using the CPYMWTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWTN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWTN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWTN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWTN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYEWTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 1 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.133 CPYPWTRN, CPYMWTRN, CPYEWTRN

Memory copy, writes unprivileged, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTRN, then CPYMWTRN, and then CPYEWTRN.

CPYPWTRN performs some preconditioning of the arguments suitable for using the CPYMWTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWTRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWTRN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWTRN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWTRN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYEWTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 0 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.134 CPYPWTWN, CPYMWTWN, CPYEWTWN

Memory copy, writes unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTWN, then CPYMWTWN, and then CPYEWTWN.

CPYPWTWN performs some preconditioning of the arguments suitable for using the CPYMWTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWTWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWTWN:

- If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs<55:0> > Xd<55:0>) and (Xd<55:0> + saturated copy size) > Xs<55:0>, then the direction is forward.
  If (Xs<55:0> < Xd<55:0>) and (Xs<55:0> + saturated copy size) > Xd<55:0>, then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWTWN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWTWN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds 0.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds 0.

For CPYEWTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds 0.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 1 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
constant bits(4) options = op2;
constant boolean rnontemporal = options<3> == '1';
constant boolean wnontemporal = options<2> == '1';
case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.<N,Z,C,V>;
memcpy.toaddress   = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();

constant boolean rprivileged = (if options<1> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);
constant boolean wprivileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                                else PSTATE.EL != EL0);

constant AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
constant AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
        CheckMemCpyParams(memcpy, options);

    integer copied;
    boolean iswrite;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    boolean fault = FALSE;
    MOPSBlockSize B = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;

    else
        while memcpy.stagecpysize > 0 && !fault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);
            assert B <= memcpy.stagecpysize;

            if memcpy.forward then
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                        memcpy.fromaddress,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.135 CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum

This instruction performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x04C11DB7` is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC

(FEAT_CRC32)

31	30	29	28							21	20				16	15			13	12	11		10	9					5	4					0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	0		0	sz			Rn						Rd					
		S																		C															

Encoding for the CRC32B variant

Applies when `(sf == 0 && sz == 00)`

```
CRC32B  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32H variant

Applies when `(sf == 0 && sz == 01)`

```
CRC32H  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32W variant

Applies when `(sf == 0 && sz == 10)`

```
CRC32W  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32X variant

Applies when `(sf == 1 && sz == 11)`

```
CRC32X  <Wd>, <Wn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if sf == '1' && sz != '11' then EndOfDecode(Decode_UNDEF);
if sf == '0' && sz == '11' then EndOfDecode(Decode_UNDEF);
constant integer size = 8 << UInt(sz);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose accumulator output register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose accumulator input register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the general-purpose data source register, encoded in the “Rm” field.

<Xm>

Is the 64-bit name of the general-purpose data source register, encoded in the “Rm” field.

Operation

```
constant bits(32)      acc     = X[n, 32];     // accumulator
constant bits(size)    val     = X[m, size];   // input value
constant bits(32)      poly    = \texttt{0x04C11DB7}<31:0>;

constant bits(32+size) tempacc = BitReverse(acc):Zeros(size);
constant bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.136 CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32C checksum

This instruction performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x1EDC6F41` is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC

(FEAT_CRC32)

Encoding for the CRC32CB variant

Applies when `(sf == 0 && sz == 00)`

```
CRC32CB  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CH variant

Applies when `(sf == 0 && sz == 01)`

```
CRC32CH  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CW variant

Applies when `(sf == 0 && sz == 10)`

```
CRC32CW  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CX variant

Applies when `(sf == 1 && sz == 11)`

```
CRC32CX  <Wd>, <Wn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if sf == '1' && sz != '11' then EndOfDecode(Decode_UNDEF);
if sf == '0' && sz == '11' then EndOfDecode(Decode_UNDEF);
constant integer size = 8 << UInt(sz);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose accumulator output register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose accumulator input register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the general-purpose data source register, encoded in the “Rm” field.

<Xm>

Is the 64-bit name of the general-purpose data source register, encoded in the “Rm” field.

Operation

```
constant bits(32)      acc     = X[n, 32];     // accumulator
constant bits(size)    val     = X[m, size];   // input value
constant bits(32)      poly    = \texttt{0x1EDC6F41}<31:0>;

constant bits(32+size) tempacc = BitReverse(acc):Zeros(size);
constant bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.137 CSDB

Consumption of speculative data barrier

This instruction is a memory barrier that controls speculative execution arising from data value prediction. For more information and details of the semantics, see Consumption of Speculative Data Barrier (CSDB).

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	1 0 0 (op2)	1 1 1 1 1

Encoding

```
CSDB
```

Decode for this encoding

```
// Empty.
```

Operation

```
ConsumptionOfSpeculativeDataBarrier();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.138 CSEL

Conditional select

This instruction writes the value of the first source register to the destination register if the condition is TRUE. If the condition is FALSE, it writes the value of the second source register to the destination register.

31	30	29	28 … 21	20 … 16	15 … 12	11	10	9 … 5	4 … 0
sf	0	0	1 1 0 1 0 1 0 0	Rm	cond	0	0	Rn	Rd
	op	S					o2		

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSEL  <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSEL  <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
bits(datasize) result;
if ConditionHolds(condition) then
    result = X[n, datasize];
else
    result = X[m, datasize];

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.139 CSET

Conditional set

This instruction sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSET  <Wd>, <invcond>
```

is equivalent to

```
CSINC  <Wd>, WZR, WZR, <cond>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSET  <Xd>, <invcond>
```

is equivalent to

```
CSINC  <Xd>, XZR, XZR, <cond>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational Information

The description of CSINC gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.140 CSETM

Conditional set mask

This instruction sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSETM  <Wd>, <invcond>
```

is equivalent to

```
CSINV  <Wd>, WZR, WZR, <cond>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSETM  <Xd>, <invcond>
```

is equivalent to

```
CSINV  <Xd>, XZR, XZR, <cond>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational Information

The description of CSINV gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.141 CSINC

Conditional select increment

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC and CSET.

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	0	0	Rm					cond					0	1	Rn					Rd				
	op	S																				o2										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSINC  <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSINC  <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CINC	Rm != '11111' && Rn != '11111' && !(cond IN {'111x'}) && Rn == Rm
CSET	Rm == '11111' && Rn == '11111' && !(cond IN {'111x'})

Operation

```
bits(datasize) result;
if ConditionHolds(condition) then
    result = X[n, datasize];
else
    result = X[m, datasize] + 1;

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.142 CSINV

Conditional select invert

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV and CSETM.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSINV  <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSINV  <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CINV	Rm != '11111' && Rn != '11111' && !(cond IN {'111x'}) && Rn == Rm
CSETM	Rm == '11111' && Rn == '11111' && !(cond IN {'111x'})

Operation

```
bits(datasize) result;
if ConditionHolds(condition) then
    result = X[n, datasize];
else
    result = NOT(X[m, datasize]);

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.143 CSNEG

Conditional select negation

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSNEG  <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSNEG  <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CNEG	`!(cond IN {'111x'}) && Rn == Rm`

Operation

```
bits(datasize) result;
if ConditionHolds(condition) then
    result = X[n, datasize];
else
    result = NOT(X[m, datasize]) + 1;

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.144 CTZ

Count trailing zeros

This instruction counts the number of consecutive binary zero bits, starting from the least significant bit in the source register, and places the count in the destination register.

Integer

(FEAT_CSSC)

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CTZ  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CTZ  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer datasize = 32 << UInt(sf);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = CountLeadingZeroBits(BitReverse(operand1));
X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.145 DC

Data cache operation

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding

```
DC  <dc_op>, <Xt>
```

is equivalent to

```
SYS  #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_DC`.

Assembler Symbols

<dc_op>

Is a DC operation name, as listed for the DC system instruction group, encoded in “op1:CRm:op2”:

op1	CRm	op2	<dc_op>	Architectural Feature
000	0110	001	IVAC	-
000	0110	010	ISW	-
000	0110	011	IGVAC	FEAT_MTE2
000	0110	100	IGSW	FEAT_MTE2
000	0110	101	IGDVAC	FEAT_MTE2
000	0110	110	IGDSW	FEAT_MTE2
000	1010	010	CSW	-
000	1010	100	CGSW	FEAT_MTE2
000	1010	110	CGDSW	FEAT_MTE2
000	1110	010	CISW	-
000	1110	100	CIGSW	FEAT_MTE2
000	1110	110	CIGDSW	FEAT_MTE2
000	1111	001	CIVAPS	FEAT_PoPS
000	1111	101	CIGDVAPS	FEAT_PoPS && FEAT_MTE2
011	0100	001	ZVA	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRm	op2	<dc_op>	Architectural Feature
011	0100	011	GVA	FEAT_MTE
011	0100	100	GZVA	FEAT_MTE
011	1010	001	CVAC	-
011	1010	011	CGVAC	FEAT_MTE
011	1010	101	CGDVAC	FEAT_MTE
011	1011	000	CVAOC	FEAT_OCCMO
011	1011	001	CVAU	-
011	1011	111	CGDVAOC	FEAT_OCCMO && FEAT_MTE
011	1100	001	CVAP	FEAT_DPB
011	1100	011	CGVAP	FEAT_MTE
011	1100	101	CGDVAP	FEAT_MTE
011	1101	001	CVADP	FEAT_DPB2
011	1101	011	CGVADP	FEAT_MTE
011	1101	101	CGDVADP	FEAT_MTE
011	1110	001	CIVAC	-
011	1110	011	CIGVAC	FEAT_MTE
011	1110	101	CIGDVAC	FEAT_MTE
011	1111	000	CIVAOC	FEAT_OCCMO
011	1111	111	CIGDVAOC	FEAT_OCCMO && FEAT_MTE
100	1110	000	CIPAE	FEAT_MEC
100	1110	111	CIGDPAE	FEAT_MEC && FEAT_MTE2
110	1110	001	CIPAPA	FEAT_RME
110	1110	101	CIGDPAPA	FEAT_RME && FEAT_MTE2

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.146 DCPS1

Debug change PE state to EL1

This instruction, when executed in Debug state:

- If executed at EL0, changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target Exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is always UNDEFINED in Non-debug state.

This instruction is UNDEFINED at EL0 if EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE == 1.

For more information on the operation of the DCPS<n> instructions, see DCPS.

Encoding

```
DCPS1  {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm>

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the “imm16” field.

Operation

```
if !Halted() then UNDEFINED;
DCPSInstruction(EL1);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.147 DCPS2

Debug change PE state to EL2

This instruction, when executed in Debug state:

- If executed at EL0 or EL1, changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target Exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an Exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is always UNDEFINED in Non-debug state.

This instruction is UNDEFINED at the following Exception levels:

- All Exception levels if EL2 is not implemented.
- At EL0 and EL1 if EL2 is disabled in the current Security state.

For more information on the operation of the DCPS<n> instructions, see DCPS.

Encoding

```
DCPS2  {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm>

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the “imm16” field.

Operation

```
if !Halted() then UNDEFINED;
DCPSInstruction(EL2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.148 DCPS3

Debug change PE state to EL3

This instruction, when executed in Debug state:

- If executed at EL3, selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target Exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

- ELR_EL3 becomes UNKNOWN.
- SPSR_EL3 becomes UNKNOWN.
- ESR_EL3 becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_EL3.EE.

This instruction is always UNDEFINED in Non-debug state.

This instruction is UNDEFINED at all Exception levels if either:

- EDSCR.SDD == 1.
- EL3 is not implemented.

For more information on the operation of the DCPS<n> instructions, see DCPS.

Encoding

```
DCPS3  {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm>

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the “imm16” field.

Operation

```
if !Halted() then UNDEFINED;
DCPSInstruction(EL3);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.149 DGH

Data gathering hint

This instruction is a hint instruction that indicates that it is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

System

(FEAT_DGH)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	1 1 0 (op2)	1 1 1 1 1

Encoding

```
DGH
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DGH) then EndOfDecode(Decode_NOP);
```

Operation

```
Hint_DGH();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.150 DMB

Data memory barrier

This instruction is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier.

31 29	28 26	25 12	11 8	7	6 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	1	0 1 (opc)	1 1 1 1 1 (Rt)

Encoding

```
DMB  (<option>|#<imm>)
```

Decode for this encoding

```
MBReqDomain domain;
MBReqTypes types;
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
```

Assembler Symbols

<option>

Specifies the limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = `0b1111`.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1110`.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1101`.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b1011`.

ISHST Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1010`.

ISHLD Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1001`.

NSH Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = `0b0111`.

NSHST Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0110`.

NSHLD Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0101`.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b0011`.

OSHST Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0010`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSHLD Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0001`.

All other encodings of "CRm" that are not listed are reserved and can be encoded using the `#<imm>` syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

CRm	<option>
0001	OSHLD
0010	OSHST
0011	OSH
0101	NSHLD
0110	NSHST
0111	NSH
xx00	RESERVED
1001	ISHLD
1010	ISHST
1011	ISH
1101	LD
1110	ST
1111	SY

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
DataMemoryBarrier(domain, types);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.151 DRPS

Debug restore PE state

This instruction restores PSTATE from the SPSR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

This instruction is UNDEFINED in Non-debug state.

This instruction is UNDEFINED at EL0.

For more information on the operation of DRPS, see DRPS.

31 29	28 26	25	24 21	20 16	15 10	9 5	4 0
1 1 0	1 0 1	1	0 1 0 1	1 1 1 1 1	0 0 0 0 0 0	1 1 1 1 1	0 0 0 0 0
			opc	op2	op3	Rn	op4

Encoding

```
DRPS
```

Decode for this encoding

```
// Empty.
```

Operation

```
if !Halted() || PSTATE.EL == EL0 then UNDEFINED;
DRPSInstruction();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.152 DSB

Data synchronization barrier

This instruction is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier.

This instruction is used by the aliases PSSBB and SSBB.

It has encodings from 2 classes: Memory barrier and Memory nXS barrier

Memory barrier

31–29	28–26	25–12	11–8	7	6–5	4–0
110	101	0100000110011	CRm	1	00 (opc)	11111 (Rt)

Encoding

```
DSB  (<option>|#<imm>)
```

Decode for this encoding

```
boolean nXS = FALSE;

DSBAlias alias;
case CRm of
    when '0000' alias = DSBAlias_SSBB;
    when '0100' alias = DSBAlias_PSSBB;
    otherwise   alias = DSBAlias_DSB;

MBReqDomain domain;
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

MBReqTypes types;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
```

Memory nXS barrier

(FEAT_XS)

31–29	28–26	25–12	11–10	9–8	7–5	4–0
110	101	0100000110011	imm2	10	001 (op2)	11111 (Rt)

Encoding

```
DSB  <option>nXS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_XS) then EndOfDecode(Decode_UNDEF);
constant MBReqTypes types = MBReqTypes_All;
boolean nXS = TRUE;
constant DSBAlias alias = DSBAlias_DSB;
MBReqDomain domain;

case imm2 of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;
```

Assembler Symbols

<option>

For the “Memory barrier” variant: specifies the limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = `0b1111`.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1110`.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1101`.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b1011`.

ISHST Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1010`.

ISHLD Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1001`.

NSH Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = `0b0111`.

NSHST Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0110`.

NSHLD Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0101`.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b0011`.

OSHST Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0010`.

OSHLD Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0001`.

All other encodings of “CRm” that are not listed, other than the values `0b0000` and `0b0100`, are reserved and can be encoded using the `#<imm>` syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

Note

The value `0b0000` is used to encode `SSBB` and the value `0b0100` is used to encode `PSSBB`.

CRm	<option>
0001	OSHLD

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRm	<option>
0010	OSHST
0011	OSH
0101	NSHLD
0110	NSHST
0111	NSH
1001	ISHLD
1010	ISHST
1011	ISH
1x00	RESERVED
1101	LD
1110	ST
1111	SY

For the “Memory nXS barrier” variant: specifies the limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as imm2 = `0b11`.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as imm2 = `0b10`.

NSH Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as imm2 = `0b01`.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as imm2 = `0b00`.

imm2	<option>
00	OSH
01	NSH
10	ISH
11	SY

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field.

Alias Conditions

Alias	Is preferred when
PSSBB	CRm == '0100'
SSBB	CRm == '0000'

Operation

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
case alias of
    when DSBAlias_SSBB
        SpeculativeStoreBypassBarrierToVA();
    when DSBAlias_PSSBB
        SpeculativeStoreBypassBarrierToPA();
    when DSBAlias_DSB
        if !nXS && IsFeatureImplemented(FEAT_XS) then
            nXS = PSTATE.EL IN {EL0, EL1} && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1';
        DataSynchronizationBarrier(domain, types, nXS);
    otherwise
        Unreachable();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.153 DVP

Data value prediction restriction by context

This instruction prevents data value predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see DVP RCTX, Data Value Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 1	1 0 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
DVP  RCTX, <Xt>
```

is equivalent to

```
SYS  #3, C7, C3, #5, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.154 EON (shifted register)

Bitwise exclusive-OR NOT (shifted register)

This instruction performs a bitwise exclusive-OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 0 (opc)	0 1 0 1 0	shift	1 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

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

X[d, datasize] = operand1 EOR NOT(operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.155 EOR (immediate)

Bitwise exclusive-OR (immediate)

This instruction performs a bitwise exclusive-OR of a register value and an immediate value, and writes the result to the destination register.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 0 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
EOR  <Wd|WSP>, <Wn>, #<imm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
EOR  <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”.

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

constant bits(datasize) result = operand1 EOR operand2;

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.156 EOR (shifted register)

Bitwise exclusive-OR (shifted register)

This instruction performs a bitwise exclusive-OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 0	0 1 0 1 0	shift	0	Rm	imm6	Rn	Rd
	opc			N				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

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

X[d, datasize] = operand1 EOR operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.157 ERET

Exception return

This instruction restores PSTATE from the SPSR, and branches to the address held in the ELR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

ERET is UNDEFINED at EL0.

31–29	28–26	25	24–21	20–16	15–12	11	10	9–5	4–0
1 1 0	1 0 1	1	0 1 0 0 (opc)	1 1 1 1 1 (op2)	0 0 0 0	0 (A)	0 (M)	1 1 1 1 1 (Rn)	0 0 0 0 0 (op4)

Encoding

```
ERET
```

Decode for this encoding

```
constant boolean pac = FALSE;
constant boolean use_key_a = TRUE;
```

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;
AArch64.CheckForERetTrap(pac, use_key_a);
constant bits(64) target = ELR_ELx[];

AArch64.ExceptionReturn(target, SPSR_ELx[]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.158 ERETAA, ERETAB

Exception return, with pointer authentication

This instruction authenticates the address in ELR, using SP as the modifier and the specified key, restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for `ERETAA`. Key B is used for `ERETAB`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to ELR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

`ERETAA` and `ERETAB` are UNDEFINED at EL0.

Integer

(FEAT_PAuth)

31						25	24			21	20				16	15				11	10	9				5	4				0
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1
									opc				op2							A				Rn					op4		

Encoding for the ERETAA variant

Applies when `(M == 0)`

```
ERETAA
```

Encoding for the ERETAB variant

Applies when `(M == 1)`

```
ERETAB
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

constant boolean pac = TRUE;
constant boolean use_key_a = (M == '0');
constant boolean auth_then_branch = TRUE;
```

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;
AArch64.CheckForERetTrap(pac, use_key_a);
bits(64) target = ELR_ELx[];
constant bits(64) modifier = SP[64];

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);

AArch64.ExceptionReturn(target, SPSR_ELx[]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.159 ESB

Error synchronization barrier

This instruction is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.

This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. For more information, see RAS PE architecture and Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100).

If FEAT_RAS is not implemented, this instruction executes as a `NOP`.

System

(FEAT_RAS)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	0 0 0 (op2)	1 1 1 1 1

Encoding

```
ESB
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then EndOfDecode(Decode_NOP);
```

Operation

```
SynchronizeErrors();
AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    AArch64.vESBOperation();
elsif IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 then
    AArch64.dESBOperation();
TakeUnmaskedSErrorInterrupts();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.160 EXTR

Extract register

This instruction extracts a register from a pair of registers.

This instruction is used by the alias ROR (immediate).

31	30 29	28 23	22	21	20 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 1	N	0	Rm	imms	Rn	Rd
	op21			o0				

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms == 0xxxxx)`

```
EXTR  <Wd>, <Wn>, <Wm>, #<lsb>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
EXTR  <Xd>, <Xn>, <Xm>, #<lsb>
```

Decode for all variants of this encoding

```
if N != sf then EndOfDecode(Decode_UNDEF);
if sf == '0' && imms<5> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant integer lsb = UInt(imms);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<lsb>

For the “32-bit” variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
`ROR (immediate)`	`Rn == Rm`

Operation

```
bits(datasize) result;
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant bits(2*datasize) concat = operand1:operand2;

result = concat<(lsb+datasize)-1:lsb>;

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.161 GCSB

Guarded Control Stack barrier

This instruction generates a GCSB effect.

If FEAT_GCS is not implemented, this instruction executes as a `NOP`.

System

(FEAT_GCS)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	0 1 1 (op2)	1 1 1 1 1

Encoding

```
GCSB  DSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then EndOfDecode(Decode_NOP);
```

Operation

```
GCSSynchronizationBarrier();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.162 GCSPOPCX

Guarded Control Stack pop and compare exception return record

This instruction loads an exception return record from the location indicated by the current Guarded control stack pointer register, compares the loaded values with the current ELR_ELx, SPSR_ELx, and LR, and increments the pointer by the size of a Guarded Control Stack exception return record.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

Encoding

```
GCSPOPCX
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is always the preferred disassembly.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.163 GCSPOPM

Guarded Control Stack pop

This instruction loads the 64-bit doubleword that is pointed to by the current Guarded Control Stack pointer, writes it to the destination register, and increments the current Guarded Control Stack pointer register by the size of a Guarded Control Stack procedure return record.

This is an alias of SYSL. This means:

- The encodings in this description are named to match the encodings of SYSL.
- The description of SYSL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	1	0 1	0 1 1	0 1 1 1	0 1 1 1	0 0 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
GCSPOPM    {<Xt>}
```

is equivalent to

```
SYSL  <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the optional general-purpose destination register, encoded in the “Rt” field. Defaults to XZR if absent.

Operation

The description of SYSL gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.164 GCSPOPX

Guarded Control Stack pop exception return record

This instruction loads an exception return record from the location indicated by the current Guarded Control Stack pointer register, checks that the record is an exception return record, and increments the pointer by the size of a Guarded Control Stack exception return record.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

Encoding

```
GCSPOPX
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is always the preferred disassembly.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.165 GCSPUSHM

Guarded Control Stack push

This instruction decrements the current Guarded Control Stack pointer register by the size of a Guarded control procedure return record and stores an entry to the Guarded Control Stack.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 1 1 1	0 0 0	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
GCSPUSHM  <Xt>
```

is equivalent to

```
SYS  #3, C7, C7, #0, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.166 GCSPUSHX

Guarded Control Stack push exception return record

This instruction decrements the current Guarded Control Stack pointer register by the size of a Guarded Control Stack exception return record and stores an exception return record to the Guarded Control Stack.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

Encoding

```
GCSPUSHX
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is always the preferred disassembly.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.167 GCSSS1

Guarded Control Stack switch stack 1

This instruction validates that the stack being switched to contains a Valid cap entry, stores an In-progress cap entry to the stack that is being switched to, and sets the current Guarded control stack pointer to the stack that is being switched to.

If the instruction generates a synchronous Data Abort exception, Watchpoint exception, GPC exception, or GCS Data Check exception, the value of GCSPR_ELx for the current Exception level is restored to the value held in the register before the instruction was executed.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 1 1 1	0 1 0	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
GCSSS1  <Xt>
```

is equivalent to

```
SYS  #3, C7, C7, #2, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.168 GCSSS2

Guarded Control Stack switch stack 2

This instruction validates that the most recent entry of the Guarded Control Stack being switched to contains an In-progress cap entry, stores a Valid cap entry to the Guarded Control Stack that is being switched from, and sets Xt to the Guarded Control Stack pointer that is being switched from.

This is an alias of SYSL. This means:

- The encodings in this description are named to match the encodings of SYSL.
- The description of SYSL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	1	0 1	0 1 1	0 1 1 1	0 1 1 1	0 1 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
GCSSS2  <Xt>
```

is equivalent to

```
SYSL  <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

Operation

The description of SYSL gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.169 GCSSTR

Guarded Control Stack store register

This instruction stores a doubleword from a register to memory. The address that is used for the store is calculated from a base register.

Integer

(FEAT_GCS)

31			28	27	26	25	24									15	14		12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	0	0	0	1	1	1	1	1	0	0	0	0	1	1	Rn		Rt	
																	opc								

Encoding

```
GCSSTR  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant bits(2) effective_el = PSTATE.EL;

if effective_el == PSTATE.EL then
    CheckGCSSTREnabled();

constant boolean privileged = effective_el != EL0;
constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp_STORE, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

Mem[address, 8, accdesc] = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.170 GCSSTTR

Guarded Control Stack store register (unprivileged)

This instruction stores a doubleword from a register to memory. The address that is used for the store is calculated from a base register.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1 and the Effective value of HCR_EL2.{NV1, NV} is not {1, 1}.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_GCS)

31–28	27	26	25	24–15	14–12	11–10	9–5	4–0
1 1 0 1	1	0	0	1 0 0 0 1 1 1 1 1 0	0 0 1	1 1	Rn	Rt
					opc			

Encoding

```
GCSSTTR  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant bits(2) effective_el = if AArch64.IsUnprivAccessPriv() then PSTATE.EL else EL0;

if effective_el == PSTATE.EL then
    CheckGCSSTREnabled();

constant boolean privileged = effective_el != EL0;
constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp_STORE, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Mem[address, 8, accdesc] = X[t, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.171 GMI

Tag mask insert

This instruction 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.

Integer

(FEAT_MTE)

Encoding

```
GMI  <Xd>, <Xn|SP>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
constant bits(64) address = if n == 31 then SP[64] else X[n, 64];
bits(64) mask = X[m, 64];
constant bits(4) tag = AArch64.AllocationTagFromAddress(address);

mask<UInt(tag)> = '1';
X[d, 64] = mask;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.172 HINT

Hint instruction

This instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	CRm	op2	1 1 1 1 1

Encoding

```
HINT  #<imm>
```

Decode for this encoding

```
SystemHintOp op;

boolean stream;
case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 110'
        if !IsFeatureImplemented(FEAT_DGH) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLRI";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfDecode(Decode_NOP);
    when '0010 000'
        if !IsFeatureImplemented(FEAT_RAS) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_ESB;
    when '0010 001'
        if !IsFeatureImplemented(FEAT_SPE) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_PSB;
    when '0010 010'
        if !IsFeatureImplemented(FEAT_TRF) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_TSB;
    when '0010 011'
        if !IsFeatureImplemented(FEAT_GCS) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_GCSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0010 110'
        if !IsFeatureImplemented(FEAT_CLRBHB) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_CLRBHB;
    when '0011 xxx'
        case op2 of
            when '000' SEE "PACIAZ";
            when '001' SEE "PACIASP";
            when '010' SEE "PACIBZ";
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '011' SEE "PACIBSP";
            when '100' SEE "AUTIAZ";
            when '101' SEE "AUTIASP";
            when '110' SEE "AUTIBZ";
            when '111' SEE "AUTIBSP";
    when '0100 xx0'
        if !IsFeatureImplemented(FEAT_BTI) then EndOfDecode(Decode_NOP);

        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
        op = SystemHintOp_BTI;
    when '0100 111' SEE "PACM";
    when '0101 000'
        if !IsFeatureImplemented(FEAT_CHK) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_CHKFEAT;
    when '0110 00x'
        if !IsFeatureImplemented(FEAT_PCDPHINT) then EndOfDecode(Decode_NOP);
        stream = op2<0> == '1';
        op = SystemHintOp_STSHH;
    otherwise EndOfDecode(Decode_NOP);
```

Assembler Symbols

<imm>

Is a 7-bit unsigned immediate, in the range 0 to 127, encoded in the “CRm:op2” field.

The encodings that are allocated to architectural hint functionality are described in the ‘Hints’ table in the ‘Index by Encoding’.

Note

For allocated encodings of “CRm:op2”:

- A disassembler will disassemble the allocated instruction, rather than the `HINT` instruction.
- An assembler may support assembly of allocated encodings using `HINT` with the corresponding `<imm>` value, but it is not required to do so.

Operation

```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_DGH
        Hint_DGH();

    when SystemHintOp_WFE
        Hint_WFE();

    when SystemHintOp_WFI
        Hint_WFI();

    when SystemHintOp_SEV
        SendEvent();

    when SystemHintOp_SEVL
        SendEventLocal();

    when SystemHintOp_ESB
        SynchronizeErrors();
        AArch64.ESBOperation();
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
            AArch64.vESBOperation();
        elsif IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 then
            AArch64.dESBOperation();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSB
        if IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_SPEv1p5) then
            constant boolean trap_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                            !IsInHost() &&
                                            (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
                                            HFGITR_EL2.PSBCSYNC == '1');
            if trap_to_el2 then
                ExceptionRecord except = ExceptionSyndrome(Exception_LDST64BTrap); // to be renamed
                except.syndrome.iss = \texttt{0x3}<24:0>;
                constant bits(64) preferred_exception_return = ThisInstrAddr(64);
                constant integer vect_offset = \texttt{0x0};
                AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);

        ProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        if IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_TRBEv1p1) then
            constant boolean trap_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                            !IsInHost() &&
                                            (!HaveEL(EL3) || SCR_EL3.FGTEn2 == '1') &&
                                            HFGITR2_EL2.TSBCSYNC == '1');
            if trap_to_el2 then
                ExceptionRecord except = ExceptionSyndrome(Exception_LDST64BTrap); // to be renamed
                except.syndrome.iss = \texttt{0x4}<24:0>;
                constant bits(64) preferred_exception_return = ThisInstrAddr(64);
                constant integer vect_offset = \texttt{0x0};
                AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);

        TraceSynchronizationBarrier();

    when SystemHintOp_GCSB
        GCSSynchronizationBarrier();

    when SystemHintOp_CHKFEAT
        X[16, 64] = AArch64.ChkFeat(X[16, 64]);

    when SystemHintOp_CSDB
        ConsumptionOfSpeculativeDataBarrier();

    when SystemHintOp_CLRBHB
        Hint_CLRBHB();

    when SystemHintOp_BTI
        SetBTypeNext('00');

    when SystemHintOp_STSHH
        Hint_StoreShared(stream);

    when SystemHintOp_NOP
        return; // Do nothing

    otherwise
        Unreachable();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.173 HLT

Halt instruction

This instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

Within a guarded memory region, while PSTATE.BTYPE != `0b00`, an `HLT` instruction that would cause entry into Debug state will not generate a Branch Target exception and will cause entry into Debug state as normal. For more information, see PSTATE.BTYPE.

Encoding

```
HLT  #<imm>
```

Decode for this encoding

```
if EDSCR.HDE == '0' || !HaltingAllowed() then
    EndOfDecode(Decode_UNDEF);
elsif IsFeatureImplemented(FEAT_BTI) then
    SetBTypeCompatible(TRUE);
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
constant boolean is_async = FALSE;
constant FaultRecord fault = NoFault();
Halt(DebugHalt_HaltInstruction, is_async, fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.174 HVC

Hypervisor call

This instruction causes an exception to EL2. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The `HVC` instruction is UNDEFINED:

- When EL3 is implemented and SCR_EL3.HCE is set to 0.
- When EL3 is not implemented and HCR_EL2.HCD is set to 1.
- When EL2 is not implemented.
- At EL1 if EL2 is not enabled in the current Security state.
- At EL0.

On executing an `HVC` instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC syndrome value `0x16`, and the value of the immediate argument.

31 29	28 26	25 24	23 21	20 5	4 2	1 0
1 1 0	1 0 1	0 0	0 0 0	imm16	0 0 0	1 0
			opc		op2	LL

Encoding

```
HVC  #<imm>
```

Decode for this encoding

```
if !HaveEL(EL2) then EndOfDecode(Decode_UNDEF);
constant bits(16) imm = imm16;
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;
if PSTATE.EL == EL1 && !EL2Enabled() then UNDEFINED;
if !HaveEL(EL3) && HCR_EL2.HCD == '1' then UNDEFINED;
if HaveEL(EL3) && SCR_EL3.HCE == '0' then UNDEFINED;

AArch64.CallHypervisor(imm);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.175 IC

Instruction cache operation

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0 (L)	0 1	op1	0 1 1 1 (CRn)	CRm	op2	Rt

Encoding

```
IC  <ic_op>{, <Xt>}
```

is equivalent to

```
SYS  #<op1>, C7, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_IC`.

Assembler Symbols

<ic_op>

Is an IC operation name, as listed for the IC system instruction pages, encoded in “op1:CRm:op2”:

op1	CRm	op2	<ic_op>
000	0001	000	IALLUIS
000	0101	000	IALLU
011	0101	001	IVAU

<Xt>

Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.176 IRG

Insert random tag

This instruction 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.

Integer

(FEAT_MTE)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	0	0	1	1 0 1	0 1 1 0	Rm	0 0 0 1 0 0	Rn	Rd
sf		S					opcode		

Encoding

```
IRG  <Xd|SP>, <Xn|SP>{, <Xm>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field. Defaults to XZR if absent.

Operation

```
constant bits(64) operand = if n == 31 then SP[64] else X[n, 64];
constant bits(64) exclude_reg = X[m, 64];
constant bits(16) exclude = exclude_reg<15:0> OR GCR_EL1.Exclude;

constant bits(4)  rtag = AArch64.ChooseTagOrZero(exclude);
constant bits(64) result = AArch64.AddressWithAllocationTag(operand, rtag);

if d == 31 then
    SP[64] = result;
else
    X[d, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.177 ISB

Instruction synchronization barrier

This instruction flushes the pipeline in the PE and is a context synchronization event. For more information, see Instruction Synchronization Barrier (ISB).

31–29	28–26	25–12	11–8	7	6–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	1	1 0 (opc)	1 1 1 1 1 (Rt)

Encoding

```
ISB  {<option>|#<imm>}
```

Decode for this encoding

```
// no additional decoding required
```

Assembler Symbols

<option>

Specifies an optional limitation on the barrier operation. Values are:

SY Full system barrier operation, encoded as CRm = `0b1111`. Can be omitted.

All other encodings of "CRm" are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

<imm>

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
InstructionSynchronizationBarrier();

if IsFeatureImplemented(FEAT_BRBE) && BRBEBranchOnISB() then
    BRBEISB();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.178 LD64B

Single-copy atomic 64-byte Load

This instruction derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers. The load starts at register `Xt`, with the data being read as `X(t+7)::X(t+6)::X(t+5)::X(t+4)::X(t+3)::X(t+2)::X(t+1)::Xt = Data[511:0]`. The data is loaded atomically and is required to be 64-byte aligned.

It is IMPLEMENTATION DEFINED which memory locations support this instruction. A memory location that supports `LD64B` also supports `ST64B`.

For more information, including about the memory types accessible and how the accesses are performed, see Single-copy atomic 64-byte load/store.

Integer

(FEAT_LS64)

Encoding

```
LD64B  <Xt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64) then EndOfDecode(Decode_UNDEF);
if Rt<4:3> == '11' || Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant boolean withstatus = FALSE;
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean tagchecked = n != 31;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LD64B.

Assembler Symbols

<Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;

constant AccessDescriptor accdesc = CreateAccDescLS64(MemOp_LOAD, withstatus, tagchecked);

if n == 31 then
    CheckSPAlignment();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = SP[64];
else
    address = X[n, 64];
data = MemLoad64B(address, accdesc);

for i = 0 to 7
    value = data<63+64*i : 64*i>;
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    X[t+i, 64] = value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.179 LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADD, STADDL.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDADD  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDADDA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDADDAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDADDL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDADD  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDADDA  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDADDAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDADDL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STADD, STADDL	A == '0' && Rt == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.180 LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADDB, STADDLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDADDB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDADDAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDADDALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDADDLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STADDB, STADDLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.181 LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword

This instruction atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAH` and `LDADDALH` load from memory with acquire semantics.
  - `LDADDLH` and `LDADDALH` store to memory with release semantics.
  - `LDADDH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADDH, STADDLH.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDADDH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDADDAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDADDALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDADDLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STADDH, STADDLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.182 LDAPR

Load-acquire RCpc register

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPR` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

It has encodings from 2 classes: Post-index and No offset

Post-index

(FEAT_LRCPC3)

31	30	29						23	22	21										10	9			5	4			0
1	x	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn				Rt		
size									L																			

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAPR  <Wt>, [<Xn|SP>], #4
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAPR  <Xt>, [<Xn|SP>], #8
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC3) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
boolean wback = TRUE;
constant boolean acquirepc = t != 31;
constant integer regsize = if size == '11' then 64 else 32;
constant integer datasize = 8 << UInt(size);
constant integer offset = 1 << UInt(size);

constant boolean tagchecked = TRUE;

boolean wb_unknown = FALSE;

if n == t && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No offset

(FEAT_LRCPC)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	1	0	1	(1) (1) (1) (1) (1)		1	1	0 0	0	0	Rn		Rt	
size				VR		A		R		Rs		o3	opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAPR  <Wt>, [<Xn|SP> {, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAPR  <Xt>, [<Xn|SP> {, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean wback = FALSE;
constant integer offset = 0;
constant boolean wb_unknown = FALSE;
constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant integer datasize = elsize;
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = X[n, 64];

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    else
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.183 LDAPRB

Load-acquire RCpc register byte

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPRB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LRCPC)

Encoding

```
LDAPRB  <Wt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(8) data;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.184 LDAPRH

Load-acquire RCpc register halfword

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPRH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LRCPC)

Encoding

```
LDAPRH  <Wt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(16) data;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.185 LDAPUR

Load-acquire RCpc register (unscaled)

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPUR` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29					24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	0	1	1	0	0	1	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAPUR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAPUR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} datasize = 8 << scale;
constant integer{} regsize = if datasize == 64 then 64 else 32;
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.186 LDAPURB

Load-acquire RCpc register byte (unscaled)

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPURB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	0	1	1	0	0	1	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
LDAPURB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} datasize = 8;
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.187 LDAPURH

Load-acquire RCpc register halfword (unscaled)

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPURH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	0	1	1	0	0	1	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
LDAPURH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} datasize = 16;
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.188 LDAPURSB

Load-acquire RCpc register signed byte (unscaled)

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPURSB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31 30	29 … 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	0 1 1 0 0 1	1 x	0	imm9	0 0	Rn	Rt
size		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDAPURSB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDAPURSB  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} datasize = 8;
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.189 LDAPURSH

Load-acquire RCpc register signed halfword (unscaled)

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPURSH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31 30	29 … 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	0 1 1 0 0 1	1 x	0	imm9	0 0	Rn	Rt
size		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDAPURSH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDAPURSH  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} datasize = 16;
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.190 LDAPURSW

Load-acquire RCpc register signed word (unscaled)

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAPURSW` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	0	0	1	1	0	0	1	1	0	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
LDAPURSW  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} datasize = 32;
constant boolean acquirepc = t != 31;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.191 LDAR

Load-acquire register

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAR` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acquire = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
bits(64) address;
constant integer dbytes = elsize DIV 8;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = SP[64];
else
    address = X[n, 64];

constant bits(elsize) data = Mem[address,  dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.192 LDARB

Load-acquire register byte

This instruction derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDARB` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Encoding

```
LDARB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) data = Mem[address,  1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.193 LDARH

Load-acquire register halfword

This instruction derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDARH` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Encoding

```
LDARH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) data = Mem[address,  2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.194 LDATXR

Load-acquire unprivileged exclusive register

This instruction derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Note

For the purposes of the Exclusives monitors, and the forward progress guarantees for Load-Exclusive and Store-Exclusive loops, `LDATXR` is equivalent to `LDXR`.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

31	30	29						23	22	21	20					16	15	14					10	9				5	4				0
1	sz	0	0	1	0	0	1	0	1	0	(1)	(1)	(1)	(1)	(1)		1	(1)	(1)	(1)	(1)	(1)		Rn					Rt				

L, Rs, o0, Rt2

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
LDATXR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
LDATXR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acqrel = t != 31;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.195 LDAXP

Load-acquire exclusive pair of registers

This instruction 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. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. 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.

If the destination registers are not both `WZR` or not both `XZR`, `LDAXP` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

31	30	29						23	22	21	20					16	15	14				10	9					5	4					0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)		1	Rt2					Rn						Rt					
									L		Rs						o0																	

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
LDAXP  <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
LDAXP  <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
constant boolean acqrel = t != 31 && t2 != 31;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDAXP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
bits(64) address;
bits(datasize) data;

constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = elsize == 64;   // When elsize is 32, the access is single-copy atomic
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, ispair, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize]  = bits(datasize) UNKNOWN;        // In this case t = t2
elsif BigEndian(accdesc.acctype) then
    X[t, elsize]  = data<elsize +: elsize>;
    X[t2, elsize] = data<0 +: elsize>;
else
    X[t, elsize]  = data<0 +: elsize>;
    X[t2, elsize] = data<elsize +: elsize>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.196 LDAXR

Load-acquire exclusive register

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAXR` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAXR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAXR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acqrel = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.197 LDAXRB

Load-acquire exclusive register byte

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAXRB` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Encoding

```
LDAXRB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(8) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, 1, accdesc);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.198 LDAXRH

Load-acquire exclusive register halfword

This instruction 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.

If the destination register is not one of `WZR` or `XZR`, `LDAXRH` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Encoding

```
LDAXRH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(16) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, 2, accdesc);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.199 LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLR, STCLRL.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDCLR  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDCLRA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDCLRAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDCLRL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDCLR  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDCLRA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDCLRAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STCLR, STCLRL	A == '0' && Rt == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.200 LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte

This instruction 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.
  - `LDCLRLB` and `LDCLRALB` store to memory with release semantics.
  - `LDCLRB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLRB, STCLRLB.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDCLRB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDCLRAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDCLRALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDCLRLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STCLRB, STCLRLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.201 LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLRH, STCLRLH.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDCLRH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDCLRAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDCLRALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDCLRLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STCLRH, STCLRLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.202 LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL

Atomic bit clear on quadword

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and stores the result back to memory. The value initially loaded from memory is returned in the same pair of registers.

- `LDCLRPA` and `LDCLRPAL` load from memory with acquire semantics.
  - `LDCLRPL` and `LDCLRPAL` store to memory with release semantics.
  - `LDCLRP` has neither acquire nor release semantics.

Integer

(FEAT_LSE128)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9			5	4			0
0	0	0	1	1	0	0	1	A	R	1	Rt2				0	0	0	1		0	0	Rn				Rt			
	S														o3		opc												

Encoding for the LDCLRP variant

Applies when `(A == 0 && R == 0)`

```
LDCLRP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDCLRPA variant

Applies when `(A == 1 && R == 0)`

```
LDCLRPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDCLRPAL variant

Applies when `(A == 1 && R == 1)`

```
LDCLRPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDCLRPL variant

Applies when `(A == 0 && R == 1)`

```
LDCLRPL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case c of
        when Constraint_UNKNOWN     rt_unknown = TRUE;     // result is UNKNOWN
        when Constraint_UNDEF       EndOfDecode(Decode_UNDEF);
        when Constraint_NOP         EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
constant bits(64) value1 = X[t, 64];
constant bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                          tagchecked, privileged, t, t2, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) comparevalue = bits(128) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64]  = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64]  = data<63:0>;
    X[t2, 64] = data<127:64>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.203 LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive-OR on word or doubleword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEOR, STEORL.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDEOR  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDEORA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDEORAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDEORL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDEOR  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDEORA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDEORAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDEORL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STEOR, STEORL	A == '0' && Rt == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.204 LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive-OR on byte

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEORB, STEORLB.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDEORB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDEORAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDEORALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDEORLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STEORB, STEORLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.205 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive-OR on halfword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEORH, STEORLH.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDEORH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDEORAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDEORALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDEORLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STEORH, STEORLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.206 LDG

Load Allocation Tag

This instruction loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag Granule.

Integer

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	0	1	1	imm9		0	0	Rn		Rt	
						opc					op2					

Encoding

```
LDG  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;
bits(4) tag;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant boolean stzgm = FALSE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_LOAD, stzgm, t);

address = AddressAdd(address, offset, accdesc);
address = Align(address, TAG_GRANULE);

tag = AArch64.MemTag[address, accdesc];
X[t, 64] = AArch64.AddressWithAllocationTag(X[t, 64], tag);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.207 LDGM

Load tag multiple

This instruction reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and writes the Allocation Tag read from address A to the destination register at 4*A<7:4>+3:4*A<7:4>. Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)

31			28	27	26	25	24	23	22	21	20									12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn		Rt	
								opc			imm9										op2					

Encoding

```
LDGM  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
constant integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
constant boolean stzgm = FALSE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_LOAD, stzgm, t);

for i = 0 to count-1
    constant bits(4) tag = AArch64.MemTag[address, accdesc];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Elem[data, index, 4] = tag;
    address = AddressIncrement(address, TAG_GRANULE, accdesc);
    index = index + 1;

X[t, 64] = data;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.208 LDIAPP

Load-Acquire RCpc ordered pair of registers

This instruction calculates an address from a base register value and an optional offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The instruction also has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- The Memory effects associated with Xt1/Wt1 are Ordered-before the Memory effects associated with Xt2/Wt2.
- If the destination registers are not both `WZR` or not both `XZR`, `LDIAPP` loads from memory with Acquire semantics.
- 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.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LRCPC3)

Encoding for the 32-bit post-index variant

Applies when `(size == 10 && opc2 == 0000)`

```
LDIAPP  <Wt1>, <Wt2>, [<Xn|SP>], #8
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc2 == 0001)`

```
LDIAPP  <Wt1>, <Wt2>, [<Xn|SP>]
```

Encoding for the 64-bit post-index variant

Applies when `(size == 11 && opc2 == 0000)`

```
LDIAPP  <Xt1>, <Xt2>, [<Xn|SP>], #16
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc2 == 0001)`

```
LDIAPP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC3) then EndOfDecode(Decode_UNDEF);
constant boolean ispair = TRUE;
constant boolean postindex = opc2<0> == '0';
boolean wback = opc2<0> == '0';
```

`LDIAPP` has the same CONSTRAINED UNPREDICTABLE behavior as `LDP`. For information about this CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP and LDIAPP.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant integer{} scale = 2 + UInt(size<0>);
constant integer{} datasize = 8 << scale;
constant integer offset = if opc2<0> == '0' then (2 << scale) else 0;
constant boolean acqrel = t != 31 && t2 != 31;
constant boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;

boolean wb_unknown = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked, ispair, acqrel, t, t2);

if n == 31 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

bits(2*datasize) full_data;
full_data = Mem[address, 2*dbytes, accdesc];
if BigEndian(accdesc.acctype) then
    data2 = full_data<(datasize-1):0>;
    data1 = full_data<(2*datasize-1):datasize>;
else
    data1 = full_data<(datasize-1):0>;
    data2 = full_data<(2*datasize-1):datasize>;

if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;

X[t, datasize] = data1;
X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.209 LDLAR

Load LOAcquire register

This instruction loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDLAR` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load LOAcquire, Store LORelease and Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDLAR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDLAR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acquire = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
constant integer dbytes = elsize DIV 8;

constant AccessDescriptor accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(elsize) data = Mem[address,  dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.210 LDLARB

Load LOAcquire register byte

This instruction loads a byte from memory, zero-extends it and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDLARB` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load LOAcquire, Store LORelease and Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

Encoding

```
LDLARB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) data = Mem[address,  1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.211 LDLARH

Load LOAcquire register halfword

This instruction loads a halfword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDLARH` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load LOAcquire, Store LORelease and Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

Encoding

```
LDLARH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = t != 31;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) data = Mem[address,  2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.212 LDNP

Load pair of registers, with non-temporal hint

This instruction 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 addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDNP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = TRUE;
constant integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
bits(64) address;
constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
constant bits(2*datasize) data = Mem[address, 2 * dbytes, accdesc];
if rt_unknown then
    X[t, datasize]  = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    X[t2, datasize] = data<(datasize-1):0>;
    X[t, datasize]  = data<(2*datasize-1):datasize>;
else
    X[t, datasize]  = data<(datasize-1):0>;
    X[t2, datasize] = data<(2*datasize-1):datasize>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.213 LDP

Load pair of registers

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index 32-bit” and “Pre-index 32-bit” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Post-index 64-bit” and “Pre-index 64-bit” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Signed offset 32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “Signed offset 64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer{} scale = 2 + UInt(opc<1>);
constant integer{} datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
constant boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(2*datasize) data = Mem[address, 2 * dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    data2 = data<(datasize-1):0>;
    data1 = data<(2*datasize-1):datasize>;
else
    data1 = data<(datasize-1):0>;
    data2 = data<(2*datasize-1):datasize>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
X[t,  datasize] = data1;
X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.214 LDPSW

Load pair of registers signed word

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

Encoding

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

Encoding

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
```

Signed offset

Encoding

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDPSW.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index” and “Pre-index” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Signed offset” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer{} scale = 2 + UInt(opc<1>);
constant integer{} datasize = 32;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
constant boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(2*datasize) data = Mem[address, 2 * dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    data2 = data<(datasize-1):0>;
    data1 = data<(2*datasize-1):datasize>;
else
    data1 = data<(datasize-1):0>;
    data2 = data<(2*datasize-1):datasize>;

X[t,  64] = SignExtend(data1, 64);
X[t2, 64] = SignExtend(data2, 64);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.215 LDR (immediate)

Load register (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	1	0	imm9		0	1	Rn		Rt	
size				VR			opc										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDR  <Wt>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDR  <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	1	0	imm9		1	1	Rn		Rt	
size				VR			opc										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDR  <Wt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDR  <Xt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
1 x	1 1 1	0	0 1	0 1	imm12	Rn	Rt
size		VR		opc			

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDR  <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDR  <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant integer scale = UInt(size);
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDR (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<pimm>

For the “32-bit” variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the “imm12” field as <pimm>/4.

For the “64-bit” variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} datasize = 8 << scale;
constant integer{} regsize = if datasize == 64 then 64 else 32;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.216 LDR (literal)

Load register (literal)

This instruction 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 addressing modes, see Load/Store addressing modes.

31	30	29	27	26	25	24	23 … 5	4 … 0
0	x	0	1 1	0	0	0	imm19	Rt
opc				VR				

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDR  <Wt>, <label>
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
LDR  <Xt>, <label>
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer size = 4 << UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = FALSE;

constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<label>

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Operation

```
constant bits(64) address = PC64 + offset;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

X[t, size * 8] = Mem[address, size, accdesc];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.217 LDR (register)

Load register (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDR  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDR  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer scale = UInt(size);
constant integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

For the “32-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the “64-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer{} datasize = 8 << scale;
constant integer{} regsize = if datasize == 64 then 64 else 32;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.218 LDRAA, LDRAB

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`. Key B is used for `LDRAB`.

If the authentication passes, the PE behaves the same as for an `LDR` instruction. For information on behavior if the authentication fails, see Faulting on pointer authentication.

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 addressing modes, see Load/Store addressing modes.

PAC

(FEAT_PAuth)

Encoding for the Key A, offset variant

Applies when `(M == 0 && W == 0)`

```
LDRAA  <Xt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the Key A, pre-indexed variant

Applies when `(M == 0 && W == 1)`

```
LDRAA  <Xt>, [<Xn|SP>{, #<simm>}]!
```

Encoding for the Key B, offset variant

Applies when `(M == 1 && W == 0)`

```
LDRAB  <Xt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the Key B, pre-indexed variant

Applies when `(M == 1 && W == 1)`

```
LDRAB  <Xt>, [<Xn|SP>{, #<simm>}]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
boolean wback = W == '1';
constant boolean use_key_a = M == '0';
constant bits(10) S10 = S:imm9;
constant bits(64) offset = LSL(SignExtend(S10, 64), 3);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the “S:imm9” field as <simm>/8.

Operation

```
bits(64) address;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean auth_then_branch = TRUE;

constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);
if n == 31 then
    address = SP[64];
else
    address = X[n, 64];

if use_key_a then
    address = AuthDA(address, X[31, 64], auth_then_branch);
else
    address = AuthDB(address, X[31, 64], auth_then_branch);

if n == 31 then
    CheckSPAlignment();

address = AddressAdd(address, offset, accdesc);
X[t, 64] = Mem[address, 8, accdesc];

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.219 LDRB (immediate)

Load register byte (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size					VR			opc							

Encoding

```
LDRB  <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size					VR			opc							

Encoding

```
LDRB  <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	0	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size					VR			opc				

Encoding

```
LDRB  <Wt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(8) data = Mem[address, 1, accdesc];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.220 LDRB (register)

Load register byte (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the Extended register variant

Applies when `(option != 011)`

```
LDRB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the Shifted register variant

Applies when `(option == 011)`

```
LDRB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

Is the index shift amount, it must be `#0`, encoded in “S” as 0 if omitted, or as 1 if present.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(8) data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.221 LDRH (immediate)

Load register halfword (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size					VR			opc							

Encoding

```
LDRH  <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size					VR			opc							

Encoding

```
LDRH  <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	1	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size					VR			opc				

Encoding

```
LDRH  <Wt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the “imm12” field as <pimm>/2.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(16) data = Mem[address, 2, accdesc];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.222 LDRH (register)

Load register halfword (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15 … 13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	1	Rm	option	S	1	0	Rn	Rt
size					VR			opc									

Encoding

```
LDRH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);            // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(16) data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.223 LDRSB (immediate)

Load register signed byte (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	0	imm9	0 1	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSB  <Wt>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSB  <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	0	imm9	1 1	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSB  <Wt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSB  <Xt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
0 0	1 1 1	0	0 1	1 x	imm12	Rn	Rt
size		VR		opc			

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSB  <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSB  <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSB (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<pimm>

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(8) data = Mem[address, 1, accdesc];
X[t, regsize] = SignExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.224 LDRSB (register)

Load register signed byte (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the 32-bit with extended register offset variant

Applies when `(opc == 11 && option != 011)`

```
LDRSB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 32-bit with shifted register offset variant

Applies when `(opc == 11 && option == 011)`

```
LDRSB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Encoding for the 64-bit with extended register offset variant

Applies when `(opc == 10 && option != 011)`

```
LDRSB  <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 64-bit with shifted register offset variant

Applies when `(opc == 10 && option == 011)`

```
LDRSB  <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);            // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<extend>

Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount>

Is the index shift amount, it must be `#0`, encoded in “S” as 0 if omitted, or as 1 if present.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(8) data = Mem[address, 1, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.225 LDRSH (immediate)

Load register signed halfword (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSH  <Wt>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSH  <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSH  <Wt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSH  <Xt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
0 1	1 1 1	0	0 1	1 x	imm12	Rn	Rt
size		VR		opc			

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSH  <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSH  <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSH (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the “imm12” field as <pimm>/2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(16) data = Mem[address, 2, accdesc];
X[t, regsize] = SignExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.226 LDRSH (register)

Load register signed halfword (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSH  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);            // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
011	LSL
110	SXTW
111	SXTX

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(16) data = Mem[address, 2, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.227 LDRSW (immediate)

Load register signed word (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1	0	0	0	1 0	0	imm9	0 1	Rn	Rt
size			VR			opc					

Encoding

```
LDRSW  <Xt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1	0	0	0	1 0	0	imm9	1 1	Rn	Rt
size			VR			opc					

Encoding

```
LDRSW  <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28	27	26	25	24	23 22	21 … 10	9 … 5	4 … 0
1 0	1 1	1	0	0	1	1 0	imm12	Rn	Rt
size			VR			opc			

Encoding

```
LDRSW  <Xt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSW (immediate).

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the “imm12” field as <pimm>/4.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(32) data = Mem[address, 4, accdesc];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
X[t, 64] = SignExtend(data, 64);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.228 LDRSW (literal)

Load register signed word (literal)

This instruction 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 addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23 … 5	4 … 0
1	0	0	1	1	0	0	0	imm19	Rt
opc					VR				

Encoding

```
LDRSW  <Xt>, <label>
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = FALSE;

constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<label>

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Operation

```
constant bits(64) address = PC64 + offset;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

constant bits(32) data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.229 LDRSW (register)

Load register signed word (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 0	1 1	1	0	0	0	1 0	1	Rm	option	S	1 0	Rn	Rt
size			VR			opc							

Encoding

```
LDRSW  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 2 else 0;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(32) data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.230 LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSET, STSETL.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSET  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSETA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSETAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSETL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSET  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSETA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSETAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSETL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STSET, STSETL	A == '0' && Rt == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.231 LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSETB, STSETLB.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSETB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSETAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSETALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSETLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSETB, STSETLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.232 LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSETH, STSETLH.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSETH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSETAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSETALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSETLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSETH, STSETLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.233 LDSETP, LDSETPA, LDSETPAL, LDSETPL

Atomic bit set on quadword

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and stores the result back to memory. The value initially loaded from memory is returned in the same pair of registers.

- `LDSETPA` and `LDSETPAL` load from memory with acquire semantics.
  - `LDSETPL` and `LDSETPAL` store to memory with release semantics.
  - `LDSETP` has neither acquire nor release semantics.

Integer

(FEAT_LSE128)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9			5	4			0
0	0	0	1	1	0	0	1	A	R	1	Rt2				0	0	1	1		0	0	Rn				Rt			
	S														o3		opc												

Encoding for the LDSETP variant

Applies when `(A == 0 && R == 0)`

```
LDSETP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDSETPA variant

Applies when `(A == 1 && R == 0)`

```
LDSETPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDSETPAL variant

Applies when `(A == 1 && R == 1)`

```
LDSETPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDSETPL variant

Applies when `(A == 0 && R == 1)`

```
LDSETPL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case c of
        when Constraint_UNKNOWN     rt_unknown = TRUE;     // result is UNKNOWN
        when Constraint_UNDEF       EndOfDecode(Decode_UNDEF);
        when Constraint_NOP         EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
constant bits(64) value1 = X[t, 64];
constant bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                          tagchecked, privileged, t, t2, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) comparevalue = bits(128) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64]  = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64]  = data<63:0>;
    X[t2, 64] = data<127:64>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.234 LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAX, STSMAXL.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSMAX  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSMAXA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSMAXAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSMAXL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSMAX  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSMAXA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSMAXAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSMAXL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STSMAX, STSMAXL	A == '0' && Rt == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.235 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAXB, STSMAXLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSMAXB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSMAXAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSMAXALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSMAXLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSMAXB, STSMAXLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.236 LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAXH, STSMAXLH.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSMAXH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSMAXAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSMAXALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSMAXLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSMAXH, STSMAXLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.237 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMIN, STSMINL.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSMIN  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSMINA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSMINAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSMINL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSMIN  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSMINA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSMINAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSMINL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STSMIN, STSMINL	A == '0' && Rt == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.238 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMINB, STSMINLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSMINB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSMINAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSMINALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSMINLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSMINB, STSMINLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.239 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMINH, STSMINLH.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSMINH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSMINAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSMINALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSMINLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSMINH, STSMINLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.240 LDTADD, LDTADDA, LDTADDAL, LDTADDL

Atomic add unprivileged

This instruction 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`, `LDTADDA` and `LDTADDAL` load from memory with acquire semantics.
  - `LDTADDL` and `LDTADDAL` store to memory with release semantics.
  - `LDTADD` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

This instruction is used by the alias STTADD, STTADDL.

Integer

(FEAT_LSUI)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9			5	4			0
0	sz	0	1	1	0	0	1	A	R	1	Rs				0	0	0	0		0	1	Rn				Rt			
															o3	opc													

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && A == 0 && R == 0)`

```
LDTADD  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(sz == 0 && A == 1 && R == 0)`

```
LDTADDA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(sz == 0 && A == 1 && R == 1)`

```
LDTADDAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(sz == 0 && A == 0 && R == 1)`

```
LDTADDL  <Ws>, <Wt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && A == 0 && R == 0)`

```
LDTADD  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant

Applies when `(sz == 1 && A == 1 && R == 0)`

```
LDTADDA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(sz == 1 && A == 1 && R == 1)`

```
LDTADDAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(sz == 1 && A == 0 && R == 1)`

```
LDTADDL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 32 << UInt(sz);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STTADD, STTADDL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.241 LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL

Atomic bit clear unprivileged

This instruction 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`, `LDTCLRA` and `LDTCLRAL` load from memory with acquire semantics.
  - `LDTCLRL` and `LDTCLRAL` store to memory with release semantics.
  - `LDTCLR` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

This instruction is used by the alias STTCLR, STTCLRL.

Integer

(FEAT_LSUI)

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && A == 0 && R == 0)`

```
LDTCLR  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(sz == 0 && A == 1 && R == 0)`

```
LDTCLRA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(sz == 0 && A == 1 && R == 1)`

```
LDTCLRAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(sz == 0 && A == 0 && R == 1)`

```
LDTCLRL  <Ws>, <Wt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && A == 0 && R == 0)`

```
LDTCLR  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant

Applies when `(sz == 1 && A == 1 && R == 0)`

```
LDTCLRA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(sz == 1 && A == 1 && R == 1)`

```
LDTCLRAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(sz == 1 && A == 0 && R == 1)`

```
LDTCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 32 << UInt(sz);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STTCLR, STTCLRL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.242 LDTNP

Load unprivileged pair of registers, with non-temporal hint

This instruction calculates an address from a base register value and an immediate offset, loads two 64-bit doublewords from memory, and writes them to two registers. For information about addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Signed offset

(FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	0	0	0 0	1	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
LDTNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);

constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = TRUE;
constant integer datasize = 64;
constant bits(64) offset = LSL(SignExtend(imm7, 64), 3);
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

LDTNP has the same CONSTRAINED UNPREDICTABLE behavior as LDNP. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

Operation

```
bits(64) address;
constant integer dbytes = datasize DIV 8;
constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
constant bits(2*datasize) data = Mem[address, 2 * dbytes, accdesc];
if rt_unknown then
    X[t, datasize]  = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    X[t2, datasize] = data<(datasize-1):0>;
    X[t, datasize]  = data<(2*datasize-1):datasize>;
else
    X[t, datasize]  = data<(datasize-1):0>;
    X[t2, datasize] = data<(2*datasize-1):datasize>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.243 LDTP

Load unprivileged pair of registers

This instruction calculates an address from a base register value and an immediate offset, loads two 64-bit doublewords from memory, and writes them to two registers.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_LSUI)

Encoding

```
LDTP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

(FEAT_LSUI)

Encoding

```
LDTP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
boolean wback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	0	0	1 0	1	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
LDTP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
boolean wback = FALSE;
constant boolean postindex = FALSE;
```

LDTP has the same CONSTRAINED UNPREDICTABLE behavior as LDP. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Load/store register pair (post-indexed)” and “Load/store register pair (pre-indexed)” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Load/store register pair (offset)” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer{} scale = 2 + UInt(opc<1>);
constant integer{} datasize = 64;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
constant boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // Writeback is suppressed
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(2*datasize) data = Mem[address, 2 * dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    data2 = data<(datasize-1):0>;
    data1 = data<(2*datasize-1):datasize>;
else
    data1 = data<(datasize-1):0>;
    data2 = data<(2*datasize-1):datasize>;

X[t,  datasize] = data1;
X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.244 LDTR

Load register (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDTR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDTR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 8 << scale;
constant integer{} regsize = if datasize == 64 then 64 else 32;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.245 LDTRB

Load register byte (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	1	0	imm9	1	0	Rn	Rt
size					VR			opc							

Encoding

```
LDTRB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 8;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.246 LDTRH

Load register halfword (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	0	imm9	1	0	Rn	Rt
size					VR			opc							

Encoding

```
LDTRH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 16;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.247 LDTRSB

Load register signed byte (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	0	imm9	1 0	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDTRSB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDTRSB  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 8;
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.248 LDTRSH

Load register signed halfword (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDTRSH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDTRSH  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 16;
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.249 LDTRSW

Load register signed word (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1	0	0	0	1 0	0	imm9	1 0	Rn	Rt
size			VR			opc					

Encoding

```
LDTRSW  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 32;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.250 LDTSET, LDTSETA, LDTSETAL, LDTSETL

Atomic bit set unprivileged

This instruction 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`, `LDTSETA` and `LDTSETAL` load from memory with acquire semantics.
  - `LDTSETL` and `LDTSETAL` store to memory with release semantics.
  - `LDTSET` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

This instruction is used by the alias STTSET, STTSETL.

Integer

(FEAT_LSUI)

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && A == 0 && R == 0)`

```
LDTSET  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(sz == 0 && A == 1 && R == 0)`

```
LDTSETA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(sz == 0 && A == 1 && R == 1)`

```
LDTSETAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(sz == 0 && A == 0 && R == 1)`

```
LDTSETL  <Ws>, <Wt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && A == 0 && R == 0)`

```
LDTSET  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant

Applies when `(sz == 1 && A == 1 && R == 0)`

```
LDTSETA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(sz == 1 && A == 1 && R == 1)`

```
LDTSETAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(sz == 1 && A == 0 && R == 1)`

```
LDTSETL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 32 << UInt(sz);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STTSET, STTSETL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.251 LDTXR

Load unprivileged exclusive register

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Note

For the purposes of the Exclusives monitors, and the forward progress guarantees for Load-Exclusive and Store-Exclusive loops, `LDTXR` is equivalent to `LDXR`.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
LDTXR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
LDTXR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.252 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAX, STUMAXL.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDUMAX  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDUMAXA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDUMAXAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDUMAXL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDUMAX  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDUMAXA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDUMAXAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDUMAXL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STUMAX, STUMAXL	A == '0' && Rt == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.253 LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAXB, STUMAXLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDUMAXB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDUMAXAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDUMAXALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDUMAXLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STUMAXB, STUMAXLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.254 LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDUMAXH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDUMAXAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDUMAXALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDUMAXLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STUMAXH, STUMAXLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.255 LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMIN, STUMINL.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDUMIN  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDUMINA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDUMINAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDUMINL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDUMIN  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDUMINA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDUMINAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDUMINL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STUMIN, STUMINL	A == '0' && Rt == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.256 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMINB, STUMINLB.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDUMINB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDUMINAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDUMINALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDUMINLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STUMINB, STUMINLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.257 LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMINH, STUMINLH.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDUMINH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDUMINAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDUMINALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDUMINLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STUMINH, STUMINLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.258 LDUR

Load register (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31	30	29	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	1	0	imm9		0	0	Rn		Rt	
size				VR			opc										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDUR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDUR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 8 << scale;
constant integer{} regsize = if datasize == 64 then 64 else 32;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.259 LDURB

Load register byte (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	1	0	imm9	0	0	Rn	Rt
size					VR			opc							

Encoding

```
LDURB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 8;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.260 LDURH

Load register halfword (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1	1	0	0	0	0 1	0	imm9	0 0	Rn	Rt
size			VR			opc					

Encoding

```
LDURH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 16;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.261 LDURSB

Load register signed byte (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	0	imm9	0 0	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDURSB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDURSB  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 8;
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.262 LDURSH

Load register signed halfword (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1 x	0	imm9	0 0	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDURSH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDURSH  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 16;
constant integer{} regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.263 LDURSW

Load register signed word (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1	0	0	0	1 0	0	imm9	0 0	Rn	Rt
size			VR			opc					

Encoding

```
LDURSW  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 32;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 64] = SignExtend(data, 64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.264 LDXP

Load exclusive pair of registers

This instruction 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. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.

31	30	29						23	22	21	20					16	15	14				10	9				5	4				0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)		0	Rt2					Rn					Rt				
									L		Rs						o0															

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
LDXP  <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
LDXP  <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN     rt_unknown = TRUE;     // result is UNKNOWN
        when Constraint_UNDEF       EndOfDecode(Decode_UNDEF);
        when Constraint_NOP         EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDXP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
bits(64) address;
bits(datasize) data;

constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = elsize == 64;   // When elsize is 32, the access is single-copy atomic
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, ispair, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize]  = bits(datasize) UNKNOWN;        // In this case t = t2
elsif BigEndian(accdesc.acctype) then
    X[t, elsize]  = data<elsize +: elsize>;
    X[t2, elsize] = data<0 +: elsize>;
else
    X[t, elsize]  = data<0 +: elsize>;
    X[t2, elsize] = data<elsize +: elsize>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.265 LDXR

Load exclusive register

This instruction derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

For information about addressing modes, see Load/Store addressing modes.

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDXR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDXR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, t);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.266 LDXRB

Load exclusive register byte

This instruction derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24 23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1	0	0	0 0	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size						L		Rs	o0	Rt2		

Encoding

```
LDXRB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(8) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, 1, accdesc);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.267 LDXRH

Load exclusive register halfword

This instruction derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.

Encoding

```
LDXRH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(16) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                        privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, 2, accdesc);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.268 LSL (register)

Logical shift left (register)

This instruction shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is left-shifted.

This is an alias of LSLV. This means:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
LSL  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSLV  <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
LSL  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSLV  <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of LSLV gives the operational pseudocode for this instruction.

Operational Information

The description of LSLV gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.269 LSL (immediate)

Logical shift left (immediate)

This instruction shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 0	1 0 0 1 1 0	N	immr	!IN{x11111}	Rn	Rd
	opc				imms		

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms != 011111)`

```
LSL  <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #(-<shift>  MOD  32), #(31-<shift>)
```

and is the preferred disassembly when `UInt(imms) + 1 == UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1 && imms != 111111)`

```
LSL  <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM  <Xd>, <Xn>, #(-<shift>  MOD  64), #(63-<shift>)
```

and is the preferred disassembly when `UInt(imms) + 1 == UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<shift>

For the “32-bit” variant: is the shift amount, in the range 0 to 31.

For the “64-bit” variant: is the shift amount, in the range 0 to 63.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.270 LSLV

Logical shift left variable

This instruction shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register).

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0		0	0	Rn					Rd				
		S																				op2										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
LSLV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
LSLV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
LSL (register)	Unconditionally

Operation

```
constant bits(datasize) operand2 = X[m, datasize];

X[d, datasize] = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.271 LSR (register)

Logical shift right (register)

This instruction shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of LSRV. This means:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
LSR  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSRV  <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
LSR  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSRV  <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of LSRV gives the operational pseudocode for this instruction.

Operational Information

The description of LSRV gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.272 LSR (immediate)

Logical shift right (immediate)

This instruction shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 0	1 0 0 1 1 0	N	immr	x 1 1 1 1 1	Rn	Rd
	opc				imms		

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms == 011111)`

```
LSR  <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1 && imms == 111111)`

```
LSR  <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM  <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<shift>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, encoded in the “immr” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.273 LSRV

Logical shift right variable

This instruction shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register).

31	30	29	28	21	20	16	15	12	11	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 0 1 0		0	1	Rn		Rd	
		S								op2				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
LSRV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
LSRV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
LSR (register)	Unconditionally

Operation

```
constant bits(datasize) operand2 = X[m, datasize];

X[d, datasize] = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.274 MADD

Multiply-add

This instruction multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL.

31	30 29	28 24	23 21	20 16	15	14 10	9 5	4 0
sf	0 0	1 1 0 1 1	0 0 0	Rm	0	Ra	Rn	Rd
	op54		op31		o0			

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MADD  <Wd>, <Wn>, <Wm>, <Wa>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MADD  <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Wa>

Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xa>

Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
MUL	Ra == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant bits(datasize) operand3 = X[a, datasize];

constant integer result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.275 MADDPT

Multiply-add checked pointer

This instruction multiplies two register values, adds a third register value, and writes the result to the destination register. The intermediate product is treated as the offset.

If the operation would have generated a result where the most significant 8 bits of the result register differ from the most significant 8 bits of the base register, then the result is modified such that it is likely to be non-canonical when used as an address.

If the intermediate product cannot be correctly represented as a 64-bit two's complement value, then the result is modified such that it is likely to be non-canonical when used as an address.

Integer

(FEAT_CPA)

Encoding

```
MADDPT  <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CPA) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
constant bits(64) operand1 = X[n, 64];
constant bits(64) operand2 = X[m, 64];
constant bits(64) operand3 = X[a, 64];

bits(64) result;

constant integer product = SInt(operand1) * SInt(operand2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Signed and unsigned twos complement arithmetic are equivalent if only a
// fixed number of bits are considered.
result = operand3 + product<63:0>;

constant boolean overflow = (product != SInt(product<63:0>));
result = PointerMultiplyAddCheck(result, operand3, overflow);

X[d, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.276 MNEG

Multiply-negate

This instruction multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MNEG  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MSUB  <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MNEG  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MSUB  <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Operational Information

The description of MSUB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.277 MOV (to/from SP)

Move register value to or from SP

This instruction copies the value of a register to or from the stack pointer.

This is an alias of ADD (immediate). This means:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MOV  <Wd|WSP>, <Wn|WSP>
```

is equivalent to

```
ADD  <Wd|WSP>, <Wn|WSP>, #0
```

and is the preferred disassembly when `Rd == '11111' || Rn == '11111'`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd|SP>, <Xn|SP>
```

is equivalent to

```
ADD  <Xd|SP>, <Xn|SP>, #0
```

and is the preferred disassembly when `Rd == '11111' || Rn == '11111'`.

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of ADD (immediate) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.278 MOV (inverted wide immediate)

Move inverted wide immediate value

This instruction moves an inverted 16-bit immediate value to a register.

This is an alias of MOVN. This means:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOV  <Wd>, #<imm>
```

is equivalent to

```
MOVN  <Wd>, #<imm16>, LSL  #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00') && !IsOnes(imm16)`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd>, #<imm>
```

is equivalent to

```
MOVN  <Xd>, #<imm16>, LSL  #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

For the “32-bit” variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in “imm16:hw”, but excluding 0xffff0000 and `0x0000`ffff

For the “64-bit” variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in “imm16:hw”.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.279 MOV (wide immediate)

Move wide immediate value

This instruction moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22 21	20 5	4 0
sf	1 0	1 0 0 1 0 1	hw	imm16	Rd
	opc				

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOV  <Wd>, #<imm>
```

is equivalent to

```
MOVZ  <Wd>, #<imm16>, LSL  #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd>, #<imm>
```

is equivalent to

```
MOVZ  <Xd>, #<imm16>, LSL  #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

For the “32-bit” variant: is a 32-bit immediate which can be encoded in “imm16:hw”.

For the “64-bit” variant: is a 64-bit immediate which can be encoded in “imm16:hw”.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.280 MOV (bitmask immediate)

Move bitmask immediate value

This instruction writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 1 (opc)	1 0 0 1 0 0	N	immr	imms	1 1 1 1 1 (Rn)	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
MOV  <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR  <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when `!MoveWidePreferred(sf, N, imms, immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd|SP>, #<imm>
```

is equivalent to

```
ORR  <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when `!MoveWidePreferred(sf, N, imms, immr)`.

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<imm>

For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”, but excluding values which could be encoded by MOVZ or MOVN.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”, 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, encoded in the “Rd” field.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of ORR (immediate) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.281 MOV (register)

Move register value

This instruction copies the value in a source register to the destination register.

This is an alias of ORR (shifted register). This means:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MOV  <Wd>, <Wm>
```

is equivalent to

```
ORR  <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd>, <Xm>
```

is equivalent to

```
ORR  <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of ORR (shifted register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.282 MOVK

Move wide with keep

This instruction moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30 29	28 23	22 21	20 5	4 0
sf	1 1 (opc)	1 0 0 1 0 1	hw	imm16	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOVK  <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOVK  <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant bits(16) imm = imm16;
constant integer pos = UInt(hw) << 4;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

<shift>

For the “32-bit” variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the “hw” field as <shift>/16.

For the “64-bit” variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the “hw” field as <shift>/16.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

```
bits(datasize) result = X[d, datasize];
result<pos+15:pos> = imm;
X[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.283 MOVN

Move wide with NOT

This instruction moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate).

31	30 29	28 … 23	22 21	20 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 0 1	hw	imm16	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOVN  <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOVN  <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant bits(16) imm = imm16;
constant integer pos = UInt(hw) << 4;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

<shift>

For the “32-bit” variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the “hw” field as <shift>/16.

For the “64-bit” variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the “hw” field as <shift>/16.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Alias Conditions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Of variant	Is preferred when
MOV (inverted wide immediate)	32-bit	!(IsZero(imm16) && hw != '00') && !IsOnes(imm16)
MOV (inverted wide immediate)	64-bit	!(IsZero(imm16) && hw != '00')

Operation

```
bits(datasize) result = Zeros(datasize);
result<pos+15:pos> = imm;
X[d, datasize] = NOT(result);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.284 MOVZ

Move wide with zero

This instruction moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate).

31	30 29	28 … 23	22 21	20 … 5	4 … 0
sf	1 0 (opc)	1 0 0 1 0 1	hw	imm16	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOVZ  <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOVZ  <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant bits(16) imm = imm16;
constant integer pos = UInt(hw) << 4;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

<shift>

For the “32-bit” variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the “hw” field as <shift>/16.

For the “64-bit” variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the “hw” field as <shift>/16.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Alias Conditions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (wide immediate)	!(IsZero(imm16) && hw != '00')

Operation

```
bits(datasize) result = Zeros(datasize);
result<pos+15:pos> = imm;
X[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.285 MRRS

Move System register to two adjacent general-purpose registers

This instruction allows the PE to read an AArch64 128-bit System register into two adjacent 64-bit general-purpose registers.

System

(FEAT_SYSREG128)

31–29	28–26	25–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 1	1	1	o0	op1	CRn	CRm	op2	Rt
			L							

Encoding

```
MRRS  <Xt>, <Xt+1>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSREG128) then EndOfDecode(Decode_UNDEF);
if Rt<0> == '1' then EndOfDecode(Decode_UNDEF);

constant integer t       = UInt(Rt);
constant integer t2      = UInt(Rt+1);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '1' : o0;
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the first general-purpose destination register, encoded in the “Rt” field.

<Xt+1>

Is the 64-bit name of the second general-purpose destination register, encoded as “Rt” +1.

<systemreg>

Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

<op0>

Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Operation

```
AArch64.SysRegRead128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.286 MRS

Move System register to general-purpose register

This instruction allows the PE to read an AArch64 System register into a general-purpose register.

31–29	28–26	25–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	1	1	o0	op1	CRn	CRm	op2	Rt
			L							

Encoding

```
MRS  <Xt>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

Decode for this encoding

```
constant integer t       = UInt(Rt);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '1' : o0;
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<systemreg>

Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

The System register names are defined in AArch64 System Register Descriptions.

<op0>

Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Operation

```
AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.287 MSR (immediate)

Move immediate value to special register

This instruction moves an immediate value to selected bits of the PSTATE. For more information, see Process state, PSTATE.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If FEAT_SSBS is implemented, PSTATE.SSBS.
- If FEAT_PAN is implemented, PSTATE.PAN.
- If FEAT_UAO is implemented, PSTATE.UAO.
- If FEAT_DIT is implemented, PSTATE.DIT.
- If FEAT_MTE is implemented, PSTATE.TCO.
- If FEAT_NMI is implemented, PSTATE.ALLINT.
- If FEAT_SME is implemented, PSTATE.SM and PSTATE.ZA.
- If FEAT_EBEP is implemented, PSTATE.PM.

If FEAT_MTE is implemented and FEAT_MTE2 is not implemented, it is IMPLEMENTATION DEFINED whether writes to PSTATE.TCO by this instruction are ignored.

This instruction is used by the aliases SMSTART and SMSTOP.

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	op1	0 1 0 0	CRm	op2	1 1 1 1 1 (Rt)

Encoding

Applies when `(!(op1 == 000 && op2 IN {00x, 010}))`

```
MSR  <pstatefield>, #<imm>
```

Decode for this encoding

```
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
    when '00x'
        min_EL = EL1;
    when '010'
        min_EL = EL1;
    when '011'
        min_EL = EL0;
    when '100'
        min_EL = EL2;
    when '101'
        if !IsFeatureImplemented(FEAT_VHE) then EndOfDecode(Decode_UNDEF);
        min_EL = EL2;
    when '110'
        min_EL = EL3;
    when '111'
        min_EL = EL1;
        need_secure = TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(4) operand = CRm;
PSTATEField field;
case op1:op2 of
    when '000 011'
        if !IsFeatureImplemented(FEAT_UAO) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_UAO;
    when '000 100'
        if !IsFeatureImplemented(FEAT_PAN) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '001 000'
        case CRm of
            when '000x'
                if !IsFeatureImplemented(FEAT_NMI) then EndOfDecode(Decode_UNDEF);
                field = PSTATEField_ALLINT;
            when '001x'
                if !IsFeatureImplemented(FEAT_EBEP) then EndOfDecode(Decode_UNDEF);
                field = PSTATEField_PM;
            otherwise
                EndOfDecode(Decode_UNDEF);
    when '011 010'
        if !IsFeatureImplemented(FEAT_DIT) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_DIT;
    when '011 011'
        case CRm of
            when '001x'
                if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF);
                field = PSTATEField_SVCRSM;
            when '010x'
                if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF);
                field = PSTATEField_SVCRZA;
            when '011x'
                if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF);
                field = PSTATEField_SVCRSMZA;
            otherwise
                EndOfDecode(Decode_UNDEF);
    when '011 100'
        if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_TCO;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    when '011 001'
        if !IsFeatureImplemented(FEAT_SSBS) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_SSBS;
    otherwise      EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<pstatefield>

Is a PSTATE field name. For the MSR instruction, this is encoded in “CRm:op1:op2”:

CRm	op1	op2	<pstatefield>	Architectural Feature
000x	001	000	ALLINT	FEAT_NMI
000x	011	011	RESERVED	-
001x	001	000	PM	FEAT_EBEP
001x	011	011	SVCRSM	FEAT_SME
010x	011	011	SVCRZA	FEAT_SME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRm	op1	op2	<pstatefield>	Architectural Feature
01xx	001	000	RESERVED	-
011x	011	011	SVCRSMZA	FEAT_SME
xxxx	000	011	UAO	FEAT_UAO
xxxx	000	100	PAN	FEAT_PAN
xxxx	000	101	SPSel	-
xxxx	000	11x	RESERVED	-
xxxx	001	001	RESERVED	-
xxxx	001	01x	RESERVED	-
xxxx	001	1xx	RESERVED	-
xxxx	010	xxx	RESERVED	-
xxxx	011	000	RESERVED	-
xxxx	011	001	SSBS	FEAT_SSBS
xxxx	011	010	DIT	FEAT_DIT
xxxx	011	100	TCO	FEAT_MTE
xxxx	011	101	RESERVED	-
xxxx	011	110	DAIFSet	-
xxxx	011	111	DAIFClr	-
xxxx	1xx	xxx	RESERVED	-
1xxx	001	000	RESERVED	-
1xxx	011	011	RESERVED	-

For the SMSTART and SMSTOP aliases, this is encoded in “CRm<2:1>”, where `0b01` specifies SVCRSM, `0b10` specifies SVCRZA, and `0b11` specifies SVCRSMZA.

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field. Restricted to the range 0 to 1, encoded in “CRm<0>”, when <pstatefield> is ALLINT, PM, SVCRSM, SVCRSMZA, or SVCRZA.

Alias Conditions

Alias	Is preferred when
SMSTART	op1 == '011' && CRm IN {'0xx1'} && op2 == '011'
SMSTOP	op1 == '011' && CRm IN {'0xx0'} && op2 == '011'

Operation

```
if UInt(PSTATE.EL) < UInt(min_EL) then UNDEFINED;
if need_secure && CurrentSecurityState() != SS_Secure then UNDEFINED;

case field of
    when PSTATEField_SSBS
        PSTATE.SSBS = operand<0>;
    when PSTATEField_SP
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PSTATE.SP = operand<0>;
    when PSTATEField_DAIFSet
        AArch64.CheckDAIFAccess(PSTATEField_DAIFSet);
        PSTATE.D = PSTATE.D OR operand<3>;
        PSTATE.A = PSTATE.A OR operand<2>;
        PSTATE.I = PSTATE.I OR operand<1>;
        PSTATE.F = PSTATE.F OR operand<0>;
    when PSTATEField_DAIFClr
        AArch64.CheckDAIFAccess(PSTATEField_DAIFClr);
        PSTATE.D = PSTATE.D AND NOT(operand<3>);
        PSTATE.A = PSTATE.A AND NOT(operand<2>);
        PSTATE.I = PSTATE.I AND NOT(operand<1>);
        PSTATE.F = PSTATE.F AND NOT(operand<0>);
    when PSTATEField_PAN
        PSTATE.PAN = operand<0>;
    when PSTATEField_UAO
        PSTATE.UAO = operand<0>;
    when PSTATEField_DIT
        PSTATE.DIT = operand<0>;
    when PSTATEField_TCO
        PSTATE.TCO = operand<0>;
    when PSTATEField_ALLINT
        if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() &&
              HCRX_EL2.TALLINT == '1' && operand<0> == '1') then
            AArch64.SystemAccessTrap(EL2, \texttt{0x18});
        PSTATE.ALLINT = operand<0>;
    when PSTATEField_SVCRSM
        CheckSMEAccess();
        SetPSTATE_SM(operand<0>);
    when PSTATEField_SVCRZA
        CheckSMEAccess();
        SetPSTATE_ZA(operand<0>);
    when PSTATEField_SVCRSMZA
        CheckSMEAccess();
        SetPSTATE_SM(operand<0>);
        SetPSTATE_ZA(operand<0>);
    when PSTATEField_PM
        PSTATE.PM = operand<0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.288 MSR (register)

Move general-purpose register to System register

This instruction allows the PE to write an AArch64 System register from a general-purpose register.

31–29	28–26	25–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	1	o0	op1	CRn	CRm	op2	Rt
			L							

Encoding

```
MSR  (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>
```

Decode for this encoding

```
constant integer t       = UInt(Rt);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '1' : o0;
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<systemreg>

Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

The System register names are defined in AArch64 System Register Descriptions.

<op0>

Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

```
AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.289 MSRR

Move two adjacent general-purpose registers to System register

This instruction allows the PE to write an AArch64 128-bit System register from two adjacent 64-bit general-purpose registers.

System

(FEAT_SYSREG128)

31 29	28 26	25 22	21	20	19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 1	0	1	o0	op1	CRn	CRm	op2	Rt
			L							

Encoding

```
MSRR  (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>, <Xt+1>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSREG128) then EndOfDecode(Decode_UNDEF);
if Rt<0> == '1' then EndOfDecode(Decode_UNDEF);

constant integer t       = UInt(Rt);
constant integer t2      = UInt(Rt+1);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '1' : o0;
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<systemreg>

Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

<op0>

Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

<Xt>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rt” field.

<Xt+1>

Is the 64-bit name of the second general-purpose source register, encoded as “Rt” +1.

Operation

```
AArch64.SysRegWrite128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.290 MSUB

Multiply-subtract

This instruction multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG.

31	30 29	28 24	23 21	20 16	15	14 10	9 5	4 0
sf	0 0	1 1 0 1 1	0 0 0	Rm	1	Ra	Rn	Rd
	op54		op31		o0			

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MSUB  <Wd>, <Wn>, <Wm>, <Wa>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MSUB  <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Wa>

Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xa>

Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
MNEG	Ra == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant bits(datasize) operand3 = X[a, datasize];

constant integer result = UInt(operand3) - (UInt(operand1) * UInt(operand2));

X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.291 MSUBPT

Multiply-subtract checked pointer

This instruction multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register. The intermediate product is treated as the offset.

If the operation would have generated a result where the most significant 8 bits of the result register differ from the most significant 8 bits of the base register, then the result is modified such that it is likely to be non-canonical when used as an address.

If the intermediate product cannot be correctly represented as a 64-bit two's complement value, then the result is modified such that it is likely to be non-canonical when used as an address.

Integer

(FEAT_CPA)

Encoding

```
MSUBPT  <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CPA) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
constant bits(64) operand1 = X[n, 64];
constant bits(64) operand2 = X[m, 64];
constant bits(64) operand3 = X[a, 64];

bits(64) result;

constant integer product = SInt(operand1) * SInt(operand2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Signed and unsigned twos complement arithmetic are equivalent if only a
// fixed number of bits are considered.
result = operand3 - product<63:0>;

constant boolean overflow = (product != SInt(product<63:0>));
result = PointerMultiplyAddCheck(result, operand3, overflow);

X[d, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.292 MUL

Multiply

This instruction multiplies two register values and writes the result to the destination register.

This is an alias of MADD. This means:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 24	23 … 21	20 … 16	15	14 … 10	9 … 5	4 … 0
sf	0 0	1 1 0 1 1	0 0 0	Rm	0	1 1 1 1 1	Rn	Rd
	op54		op31		o0	Ra		

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MUL  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MADD  <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MUL  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MADD  <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of MADD gives the operational pseudocode for this instruction.

Operational Information

The description of MADD gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.293 MVN

Bitwise NOT

This instruction writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MVN  <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ORN  <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MVN  <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ORN  <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
01	LSR
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ORN (shifted register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.294 NEG (shifted register)

Negate (shifted register)

This instruction negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of SUB (shifted register). This means:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
NEG  <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUB  <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
NEG  <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUB  <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUB (shifted register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.295 NEGS

Negate, setting flags

This instruction negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	24	23 22	21	20	16	15	10	9	5	4	0
sf	1	1	0 1 0 1 1		shift	0	Rm		imm6		1 1 1 1 1		!=11111	
	op	S										Rn		Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
NEGS  <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS  <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
NEGS  <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS  <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (shifted register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.296 NGC

Negate with carry

This instruction negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28							21	20				16	15					10	9				5	4				0
sf	1	0	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	1	1	1	1	1	Rd				
op		S																						Rn							

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
NGC  <Wd>, <Wm>
```

is equivalent to

```
SBC  <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
NGC  <Xd>, <Xm>
```

is equivalent to

```
SBC  <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of SBC gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.297 NGCS

Negate with carry, setting flags

This instruction 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.

This is an alias of SBCS. This means:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
NGCS  <Wd>, <Wm>
```

is equivalent to

```
SBCS  <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
NGCS  <Xd>, <Xm>
```

is equivalent to

```
SBCS  <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of SBCS gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.298 NOP

No operation

This instruction 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.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	0 0 0 (op2)	1 1 1 1 1

Encoding

```
NOP
```

Decode for this encoding

```
// Empty.
```

Operation

```
return; // Do nothing
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.299 ORN (shifted register)

Bitwise OR NOT (shifted register)

This instruction performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 1	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
MVN	Rn == '11111'

Operation

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

X[d, datasize] = operand1 OR NOT(operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.300 ORR (immediate)

Bitwise OR (immediate)

This instruction performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate).

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 1 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
ORR  <Wd|WSP>, <Wn>, #<imm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ORR  <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”.

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
MOV (bitmask immediate)	Rn == '11111' && !MoveWidePreferred(sf, N, imms, immr)

Operation

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

constant bits(datasize) result = operand1 OR operand2;

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.301 ORR (shifted register)

Bitwise OR (shifted register)

This instruction performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register).

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
MOV (register)	shift == '00' && imm6 == '000000' && Rn == '11111'

Operation

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

X[d, datasize] = operand1 OR operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.302 PACDA, PACDZA

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`.

Integer

(FEAT_PAuth)

31	30	29	28							21	20				16	15	14	13	12		10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	0	Rn		Rd	
sf			S								opcode2														

Encoding for the PACDA variant

Applies when `(Z == 0)`

```
PACDA  <Xd>, <Xn|SP>
```

Encoding for the PACDZA variant

Applies when `(Z == 1 && Rn == 11111)`

```
PACDZA  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

boolean source_is_sp = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDA(X[d, 64], SP[64]);
else
    X[d, 64] = AddPACDA(X[d, 64], X[n, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.303 PACDB, PACDZB

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`.

Integer

(FEAT_PAuth)

31	30	29	28							21	20					16	15	14	13	12			10	9				5	4				0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1		0	0	Z	0	1	1		Rn					Rd				
sf																																	
			S																														
																opcode2																	

Encoding for the PACDB variant

Applies when `(Z == 0)`

```
PACDB  <Xd>, <Xn|SP>
```

Encoding for the PACDZB variant

Applies when `(Z == 1 && Rn == 11111)`

```
PACDZB  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

boolean source_is_sp = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDB(X[d, 64], SP[64]);
else
    X[d, 64] = AddPACDB(X[d, 64], X[n, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.304 PACGA

Pointer Authentication Code, using generic key

This instruction computes the Pointer Authentication Code for a 64-bit value in the first source register, using a modifier in the second source register, and the generic key. The computed Pointer Authentication Code is written to the most significant 32 bits of the destination register, and the least significant 32 bits of the destination register are set to zero.

Integer

(FEAT_PAuth)

Encoding

```
PACGA  <Xd>, <Xn>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

boolean source_is_sp = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

if m == 31 then source_is_sp = TRUE;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Rm” field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACGA(X[n, 64], SP[64]);
else
    X[d, 64] = AddPACGA(X[n, 64], X[m, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.305 PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

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`.

If FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `PACIA1716` and `PACIASP` include a second modifier that is:

- In X15, for `PACIA1716`.
- The value of PC, for `PACIASP`.

A `PACIASP` instruction has an implicit `BTI` instruction. The implicit `BTI` instruction of a `PACIASP` instruction is always compatible with PSTATE.BTYPE == `0b01` and PSTATE.BTYPE == `0b10`. Controls in SCTLR_ELx configure whether the implicit `BTI` instruction of a `PACIASP` instruction is compatible with PSTATE.BTYPE == `0b11`. For more information, see PSTATE.BTYPE.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

Encoding for the PACIA variant

Applies when `(Z == 0)`

```
PACIA  <Xd>, <Xn|SP>
```

Encoding for the PACIZA variant

Applies when `(Z == 1 && Rn == 11111)`

```
PACIZA  <Xd>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
boolean source_is_sp = FALSE;
constant boolean pacia1716 = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then EndOfDecode(Decode_UNDEF);
```

System

(FEAT_PAuth)

31																			12	11			8	7			5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	0	x		1	1	1	1	1
																				CRm						op2						

Encoding for the PACIA1716 variant

Applies when `(CRm == 0001 && op2 == 000)`

```
PACIA1716
```

Encoding for the PACIASP variant

Applies when `(CRm == 0011 && op2 == 001)`

```
PACIASP
```

Encoding for the PACIAZ variant

Applies when `(CRm == 0011 && op2 == 000)`

```
PACIAZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
integer d;
integer n;
boolean source_is_sp = FALSE;
boolean pacia1716 = FALSE;

case CRm:op2 of
    when '0011 000' // PACIAZ
        d = 30;
        n = 31;
    when '0011 001' // PACIASP
        d = 30;
        source_is_sp = TRUE;
        if IsFeatureImplemented(FEAT_BTI) then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIASP instruction.
            constant PACInstType pacinst = PACIxSP;
            SetBTypeCompatible(BTypeCompatible_PAC(pacinst));
    when '0001 000' // PACIA1716
        d = 17;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        n = 16;
        pacia1716 = TRUE;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
        X[d, 64] = AddPACIA2(X[d, 64], SP[64], PC64);
    else
        X[d, 64] = AddPACIA(X[d, 64], SP[64]);
else
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' && pacia1716 then
        X[d, 64] = AddPACIA2(X[d, 64], X[n, 64], X[15, 64]);
    else
        X[d, 64] = AddPACIA(X[d, 64], X[n, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.306 PACIA171615

Pointer Authentication Code for instruction address, using key A

This instruction computes and inserts a pointer authentication code for an instruction address, using two modifiers and key A. The address is in X17. The 64-bit value of modifier1 is the value in X16. The 64-bit value of modifier2 is the value in X15.

Integer

(FEAT_PAuth_LR)

Encoding

```
PACIA171615
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);
```

Operation

```
X[17, 64] = AddPACIA2(X[17, 64], X[16, 64], X[15, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.307 PACIASPPC

Pointer Authentication Code for return address, using key A

This instruction computes and inserts a Pointer Authentication Code for an instruction address, using two modifiers and key A.

The address is in X30.

The first modifier is in SP.

The second modifier is the value of PC.

A `PACIASPPC` instruction has an implicit `BTI` instruction. The implicit `BTI` instruction of a `PACIASPPC` instruction is always compatible with PSTATE.BTYPE == `0b01` and PSTATE.BTYPE == `0b10`. Controls in SCTLR_ELx configure whether the implicit `BTI` instruction of a `PACIASPPC` instruction is compatible with PSTATE.BTYPE == `0b11`. For more information, see PSTATE.BTYPE.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	0
sf		S									opcode2					opcode						Rn					Rd				

Encoding

```
PACIASPPC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant integer d = 30;
if IsFeatureImplemented(FEAT_BTI) then
    // Check for branch target compatibility between PSTATE.BTYPE
    // and implicit branch target of PACIxSPPC instruction.
    constant PACInstType pacinst = PACIxSPPC;
    SetBTypeCompatible(BTypeCompatible_PAC(pacinst));
```

Operation

```
X[d, 64] = AddPACIA2(X[d, 64], SP[64], PC64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.308 PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

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`.

If FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `PACIB1716` and `PACIBSP` include a second modifier that is:

- In X15, for `PACIB1716`.
- The value of PC, for `PACIBSP`.

A `PACIBSP` instruction has an implicit `BTI` instruction. The implicit `BTI` instruction of a `PACIBSP` instruction is always compatible with PSTATE.BTYPE == `0b01` and PSTATE.BTYPE == `0b10`. Controls in SCTLR_ELx configure whether the implicit `BTI` instruction of a `PACIBSP` instruction is compatible with PSTATE.BTYPE == `0b11`. For more information, see PSTATE.BTYPE.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

Encoding for the PACIB variant

Applies when `(Z == 0)`

```
PACIB  <Xd>, <Xn|SP>
```

Encoding for the PACIZB variant

Applies when `(Z == 1 && Rn == 11111)`

```
PACIZB  <Xd>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
boolean source_is_sp = FALSE;
constant boolean pacib1716 = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then EndOfDecode(Decode_UNDEF);
```

System

(FEAT_PAuth)

31																			12	11			8	7		5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	1	x	1	1	1	1	1
																				CRm					op2						

Encoding for the PACIB1716 variant

Applies when `(CRm == 0001 && op2 == 010)`

```
PACIB1716
```

Encoding for the PACIBSP variant

Applies when `(CRm == 0011 && op2 == 011)`

```
PACIBSP
```

Encoding for the PACIBZ variant

Applies when `(CRm == 0011 && op2 == 010)`

```
PACIBZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
integer d;
integer n;
boolean source_is_sp = FALSE;
boolean pacib1716 = FALSE;

case CRm:op2 of
    when '0011 010' // PACIBZ
        d = 30;
        n = 31;
    when '0011 011' // PACIBSP
        d = 30;
        source_is_sp = TRUE;
        if IsFeatureImplemented(FEAT_BTI) then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIBSP instruction.
            constant PACInstType pacinst = PACIxSP;
            SetBTypeCompatible(BTypeCompatible_PAC(pacinst));
    when '0001 010' // PACIB1716
        d = 17;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        n = 16;
        pacib1716 = TRUE;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
        X[d, 64] = AddPACIB2(X[d, 64], SP[64], PC64);
    else
        X[d, 64] = AddPACIB(X[d, 64], SP[64]);
else
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' && pacib1716 then
        X[d, 64] = AddPACIB2(X[d, 64], X[n, 64], X[15, 64]);
    else
        X[d, 64] = AddPACIB(X[d, 64], X[n, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.309 PACIB171615

Pointer Authentication Code for instruction address, using key B

This instruction computes and inserts a pointer authentication code for an instruction address, using two modifiers and key B. The address is in X17. The 64-bit value of modifier1 is the value in X16. The 64-bit value of modifier2 is the value in X15.

Integer

(FEAT_PAuth_LR)

Encoding

```
PACIB171615
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);
```

Operation

```
X[17, 64] = AddPACIB2(X[17, 64], X[16, 64], X[15, 64]);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.310 PACIBSPPC

Pointer Authentication Code for return address, using key B

This instruction computes and inserts a Pointer Authentication Code for an instruction address, using two modifiers and key B.

The address is in X30.

The first modifier is in SP.

The second modifier is the value of PC.

A `PACIBSPPC` instruction has an implicit `BTI` instruction. The implicit `BTI` instruction of a `PACIBSPPC` instruction is always compatible with PSTATE.BTYPE == `0b01` and PSTATE.BTYPE == `0b10`. Controls in SCTLR_ELx configure whether the implicit `BTI` instruction of a `PACIBSPPC` instruction is compatible with PSTATE.BTYPE == `0b11`. For more information, see PSTATE.BTYPE.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0
sf		S									opcode2					opcode						Rn					Rd				

Encoding

```
PACIBSPPC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant integer d = 30;
if IsFeatureImplemented(FEAT_BTI) then
    // Check for branch target compatibility between PSTATE.BTYPE
    // and implicit branch target of PACIxSPPC instruction.
    constant PACInstType pacinst = PACIxSPPC;
    SetBTypeCompatible(BTypeCompatible_PAC(pacinst));
```

Operation

```
X[d, 64] = AddPACIB2(X[d, 64], SP[64], PC64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.311 PACM

Pointer authentication modifier

This instruction is used to set the value of PSTATE.PACM to 1.

If FEAT_PAuth_LR is not implemented, this instruction behaves as a NOP.

System

(FEAT_PAuth_LR)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 1 0 0 (CRm)	1 1 1 (op2)	1 1 1 1 1

Encoding

```
PACM
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_NOP);
```

Operation

```
PSTATE.PACM = if IsPACMEnabled() then '1' else '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.312 PACNBIASPPC

Pointer Authentication Code for return address, using key A, not a branch target

This instruction computes and inserts a Pointer Authentication Code for an instruction address, using two modifiers and key A.

The address is in X30.

The first modifier is in SP.

The second modifier is the value of PC.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0
sf		S									opcode2					opcode						Rn					Rd				

Encoding

```
PACNBIASPPC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant integer d = 30;
```

Operation

```
X[d, 64] = AddPACIA2(X[d, 64], SP[64], PC64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.313 PACNBIBSPPC

Pointer Authentication Code for return address, using key B, not a branch target

This instruction computes and inserts a Pointer Authentication Code for an instruction address, using two modifiers and key B.

The address is in X30.

The first modifier is in SP.

The second modifier is the value of PC.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
sf		S									opcode2					opcode						Rn					Rd				

Encoding

```
PACNBIBSPPC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant integer d = 30;
```

Operation

```
X[d, 64] = AddPACIB2(X[d, 64], SP[64], PC64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.314 PRFM (immediate)

Prefetch memory (immediate)

This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from a base register value and an immediate offset. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - When FEAT_PCDPHINT is implemented, IR for intent to read on update.
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
  - When FEAT_PRFMSLC is implemented, SLC for system level cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.

Encoding

```
PRFM  (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = FALSE;
```

Assembler Symbols

<prfop>

Is the prefetch operation, encoded in “Rt”:

Rt	<prfop>	Architectural Feature
00000	PLDL1KEEP	-
00001	PLDL1STRM	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>	Architectural Feature
00010	PLDL2KEEP	-
00011	PLDL2STRM	-
00100	PLDL3KEEP	-
00101	PLDL3STRM	-
00110	PLDSLCKEEP	FEAT_PRFMSLC
00111	PLDSLCSTRM	FEAT_PRFMSLC
01000	PLIL1KEEP	-
01001	PLIL1STRM	-
01010	PLIL2KEEP	-
01011	PLIL2STRM	-
01100	PLIL3KEEP	-
01101	PLIL3STRM	-
01110	PLISLCKEEP	FEAT_PRFMSLC
01111	PLISLCSTRM	FEAT_PRFMSLC
10000	PSTL1KEEP	-
10001	PSTL1STRM	-
10010	PSTL2KEEP	-
10011	PSTL2STRM	-
10100	PSTL3KEEP	-
10101	PSTL3STRM	-
10110	PSTSLCKEEP	FEAT_PRFMSLC
10111	PSTSLCSTRM	FEAT_PRFMSLC
11000	IR	FEAT_PCDPHINT

For other encodings of the “Rt” field, use <imm5>.

<imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using `<prfop>`.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Prefetch(address, t<4:0>);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.315 PRFM (literal)

Prefetch memory (literal)

This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from the PC value and an immediate offset. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
  - When FEAT_PRFMSLC is implemented, SLC for system level cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a `PRFM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.

Encoding

```
PRFM  (<prfop>|#<imm5>), <label>
```

Decode for this encoding

```
constant integer t = UInt(Rt);

constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<prfop>

Is the prefetch operation, encoded in “Rt”:

Rt	<prfop>	Architectural Feature
00000	PLDL1KEEP	-
00001	PLDL1STRM	-
00010	PLDL2KEEP	-
00011	PLDL2STRM	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>	Architectural Feature
00100	PLDL3KEEP	-
00101	PLDL3STRM	-
00110	PLDSLCKEEP	FEAT_PRFMSLC
00111	PLDSLCSTRM	FEAT_PRFMSLC
01000	PLIL1KEEP	-
01001	PLIL1STRM	-
01010	PLIL2KEEP	-
01011	PLIL2STRM	-
01100	PLIL3KEEP	-
01101	PLIL3STRM	-
01110	PLISLCKEEP	FEAT_PRFMSLC
01111	PLISLCSTRM	FEAT_PRFMSLC
10000	PSTL1KEEP	-
10001	PSTL1STRM	-
10010	PSTL2KEEP	-
10011	PSTL2STRM	-
10100	PSTL3KEEP	-
10101	PSTL3STRM	-
10110	PSTSLCKEEP	FEAT_PRFMSLC
10111	PSTSLCSTRM	FEAT_PRFMSLC

For other encodings of the “Rt” field, use <imm5>.

<imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using `<prfop>`.

<label>

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Operation

```
constant bits(64) address = PC64 + offset;

Prefetch(address, t<4:0>);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.316 PRFM (register)

Prefetch memory (register)

This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from a base register value and an offset register value. The offset register value can optionally be shifted and extended. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
  - When FEAT_PRFMSLC is implemented, SLC for system level cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	13	12	11	10	9 … 5	4 … 0
1	1	1	1	1	0	0	0	1	0	1	Rm	x	x	S	1	0	Rn	!=11xxx
size					VR			opc				option						Rt

Encoding

```
PRFM  (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 3 else 0;
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = FALSE;
```

Assembler Symbols

<prfop>

Is the prefetch operation, encoded in “Rt”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>	Architectural Feature
00000	PLDL1KEEP	-
00001	PLDL1STRM	-
00010	PLDL2KEEP	-
00011	PLDL2STRM	-
00100	PLDL3KEEP	-
00101	PLDL3STRM	-
00110	PLDSLCKEEP	FEAT_PRFMSLC
00111	PLDSLCSTRM	FEAT_PRFMSLC
01000	PLIL1KEEP	-
01001	PLIL1STRM	-
01010	PLIL2KEEP	-
01011	PLIL2STRM	-
01100	PLIL3KEEP	-
01101	PLIL3STRM	-
01110	PLISLCKEEP	FEAT_PRFMSLC
01111	PLISLCSTRM	FEAT_PRFMSLC
10000	PSTL1KEEP	-
10001	PSTL1STRM	-
10010	PSTL2KEEP	-
10011	PSTL2STRM	-
10100	PSTL3KEEP	-
10101	PSTL3STRM	-
10110	PSTSLCKEEP	FEAT_PRFMSLC
10111	PSTSLCSTRM	FEAT_PRFMSLC

For other encodings of the “Rt” field, use <imm5>.

<imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using `<prfop>`.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

Operation

```
bits(64) address;

constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Prefetch(address, t<4:0>);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.317 PRFUM

Prefetch memory (unscaled offset)

This instruction 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 making the cache line containing the specified address available at the level of cache specified by the instruction.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
  - When FEAT_PRFMSLC is implemented, SLC for system level cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a `PRFUM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 1	1 1	1	0	0	0	1 0	0	imm9	0 0	Rn	Rt
size			VR			opc					

Encoding

```
PRFUM  (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = FALSE;
```

Assembler Symbols

<prfop>

Is the prefetch operation, encoded in “Rt”:

Rt	<prfop>	Architectural Feature
00000	PLDL1KEEP	-
00001	PLDL1STRM	-
00010	PLDL2KEEP	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>	Architectural Feature
00011	PLDL2STRM	-
00100	PLDL3KEEP	-
00101	PLDL3STRM	-
00110	PLDSLCKEEP	FEAT_PRFMSLC
00111	PLDSLCSTRM	FEAT_PRFMSLC
01000	PLIL1KEEP	-
01001	PLIL1STRM	-
01010	PLIL2KEEP	-
01011	PLIL2STRM	-
01100	PLIL3KEEP	-
01101	PLIL3STRM	-
01110	PLISLCKEEP	FEAT_PRFMSLC
01111	PLISLCSTRM	FEAT_PRFMSLC
10000	PSTL1KEEP	-
10001	PSTL1STRM	-
10010	PSTL2KEEP	-
10011	PSTL2STRM	-
10100	PSTL3KEEP	-
10101	PSTL3STRM	-
10110	PSTSLCKEEP	FEAT_PRFMSLC
10111	PSTSLCSTRM	FEAT_PRFMSLC

For other encodings of the “Rt” field, use <imm5>.

<imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using `<prfop>`.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    address = SP[64];
else
    address = X[n, 64];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
address = AddressAdd(address, offset, accdesc);

Prefetch(address, t<4:0>);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.318 PSB

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.

If FEAT_SPE is not implemented, this instruction executes as a `NOP`.

System

(FEAT_SPE)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	0 0 1 (op2)	1 1 1 1 1

Encoding

```
PSB  CSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SPE) then EndOfDecode(Decode_NOP);
```

Operation

```
if IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_SPEv1p5) then
    constant boolean trap_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                    !IsInHost() &&
                                    (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
                                    HFGITR_EL2.PSBCSYNC == '1');
    if trap_to_el2 then
        ExceptionRecord except = ExceptionSyndrome(Exception_LDST64BTrap); // to be renamed
        except.syndrome.iss = \texttt{0x3}<24:0>;
        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        constant integer vect_offset = \texttt{0x0};
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);

ProfilingSynchronizationBarrier();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.319 PSSBB

Physical speculative store bypass barrier

This instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions. For more information and details of the semantics, see Physical Speculative Store Bypass Barrier (PSSBB).

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31–29	28–26	25–12	11–8	7	6–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	0 1 0 0	1	0 0	1 1 1 1 1
			CRm		opc	Rt

Encoding

```
PSSBB
```

is equivalent to

```
DSB  #4
```

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.320 RBIT

Reverse bits

This instruction reverses the bit order in a register.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
RBIT  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
RBIT  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand = X[n, datasize];
bits(datasize) result;

for i = 0 to datasize-1
    result<(datasize-1)-i> = operand<i>;

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.321 RCWCAS, RCWCASA, RCWCASAL, RCWCASL

Read check write compare and swap doubleword in memory

This instruction reads a 64-bit doubleword from memory, and compares it against the value held in a register. If the comparison is equal, the value in a second register is conditionally written to memory. Storing back to memory is conditional on RCW Checks. If the compare fails or the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWCASA` and `RCWCASAL` load from memory with acquire semantics.
  - `RCWCASL` and `RCWCASAL` store to memory with release semantics.
  - `RCWCAS` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

31	30	29					24	23	22	21	20				16	15					10	9			5	4			0
0	0	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	0	Rn				Rt			
	S																												

Encoding for the RCWCAS variant

Applies when `(A == 0 && R == 0)`

```
RCWCAS  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCASA variant

Applies when `(A == 1 && R == 0)`

```
RCWCASA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCASAL variant

Applies when `(A == 1 && R == 1)`

```
RCWCASAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCASL variant

Applies when `(A == 0 && R == 1)`

```
RCWCASL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean soft = FALSE;

constant boolean acquire = A == '1' && t != 31;
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
constant bits(64) newdata = X[t, 64];
constant bits(64) compdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, soft, acquire, release,
                                                     tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[s, 64] = readdata;   // Return the old value when s!=31
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.322 RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL

Read check write compare and swap quadword in memory

This instruction reads a 128-bit quadword from memory, and compares it against the value held in a pair of registers. If the comparison is equal, the value in a second pair of registers is conditionally written to memory. Storing back to memory is conditional on RCW Checks. If the compare fails or the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- `RCWCASPA` and `RCWCASPAL` load from memory with acquire semantics.
  - `RCWCASPL` and `RCWCASPAL` store to memory with release semantics.
  - `RCWCASP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

31	30	29					24	23	22	21	20					16	15					10	9				5	4				0
0	0	0	1	1	0	0	1	A	R	1	Rs						0	0	0	0	1	1	Rn					Rt				
	S																															

Encoding for the RCWCASP variant

Applies when `(A == 0 && R == 0)`

```
RCWCASP  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWCASPA variant

Applies when `(A == 1 && R == 0)`

```
RCWCASPA  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWCASPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWCASPAL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWCASPL variant

Applies when `(A == 0 && R == 1)`

```
RCWCASPL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
if Rs<0> == '1' then EndOfDecode(Decode_UNDEF);
if Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean soft = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<X(s+1)>

Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>

Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<X(t+1)>

Is the 64-bit name of the second general-purpose register to be conditionally stored.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(128) newdata;
bits(128) compdata;
bits(128) readdata;
bits(4) nzcv;

constant bits(64) s1 = X[s, 64];
constant bits(64) s2 = X[s+1, 64];
constant bits(64) t1 = X[t, 64];
constant bits(64) t2 = X[t+1, 64];

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, soft, acquire, release,
                                                     tagchecked, t, t+1, s, s+1);

compdata = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newdata  = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
PSTATE.<N,Z,C,V> = nzcv;
if BigEndian(accdesc.acctype) then
    X[s, 64]   = readdata<127:64>;
    X[s+1, 64] = readdata<63:0>;
else
    X[s, 64]   = readdata<63:0>;
    X[s+1, 64] = readdata<127:64>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.323 RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL

Read check write atomic bit clear on doubleword in memory

This instruction atomically loads a 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWCLRA` and `RCWCLRAL` load from memory with acquire semantics.
  - `RCWCLRL` and `RCWCLRAL` store to memory with release semantics.
  - `RCWCLR` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

Encoding for the RCWCLR variant

Applies when `(A == 0 && R == 0)`

```
RCWCLR  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCLRA variant

Applies when `(A == 1 && R == 0)`

```
RCWCLRA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCLRAL variant

Applies when `(A == 1 && R == 1)`

```
RCWCLRAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCLRL variant

Applies when `(A == 0 && R == 1)`

```
RCWCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean soft = FALSE;

constant boolean acquire = A == '1' && t != 31;
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;

bits(64) address;
constant bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_BIC, soft, acquire, release,
                                                     tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.324 RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL

Read check write atomic bit clear on quadword in memory

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- `RCWCLRPA` and `RCWCLRPAL` load from memory with acquire semantics.
  - `RCWCLRPL` and `RCWCLRPAL` store to memory with release semantics.
  - `RCWCLRP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

Encoding for the RCWCLRP variant

Applies when `(A == 0 && R == 0)`

```
RCWCLRP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWCLRPA variant

Applies when `(A == 1 && R == 0)`

```
RCWCLRPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWCLRPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWCLRPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWCLRPL variant

Applies when `(A == 0 && R == 1)`

```
RCWCLRPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean soft = FALSE;

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_BIC, soft, acquire, release,
                                                     tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.325 RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL

Read check write software compare and swap doubleword in memory

This instruction reads a 64-bit doubleword from memory, and compares it against the value held in a register. If the comparison is equal, the value in a second register is conditionally written to memory. Storing back to memory is conditional on RCW Checks and RCWS Checks. If the compare fails, the RCW Checks fail, or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWSCASA` and `RCWSCASAL` load from memory with acquire semantics.
  - `RCWSCASL` and `RCWSCASAL` store to memory with release semantics.
  - `RCWSCAS` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

31	30	29					24	23	22	21	20				16	15					10	9			5	4			0
0	1	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	0	Rn				Rt			

S

Encoding for the RCWSCAS variant

Applies when `(A == 0 && R == 0)`

```
RCWSCAS  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCASA variant

Applies when `(A == 1 && R == 0)`

```
RCWSCASA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCASAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSCASAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCASL variant

Applies when `(A == 0 && R == 1)`

```
RCWSCASL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean soft = TRUE;

constant boolean acquire = A == '1' && t != 31;
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
constant bits(64) newdata = X[t, 64];
constant bits(64) compdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, soft, acquire, release,
                                                     tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[s, 64] = readdata;   // Return the old value when s!=31
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.326 RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL

Read check write software compare and swap quadword in memory

This instruction reads a 128-bit quadword from memory, and compares it against the value held in a pair of registers. If the comparison is equal, the value in a second pair of registers is conditionally written to memory. Storing back to memory is conditional on RCW Checks and RCWS Checks. If the compare fails, the RCW Checks fail, or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSCASPA` and `RCWSCASPAL` load from memory with acquire semantics.
  - `RCWSCASPL` and `RCWSCASPAL` store to memory with release semantics.
  - `RCWSCASP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

31	30	29					24	23	22	21	20	16	15					10	9	5	4	0
0	1	0	1	1	0	0	1	A	R	1	Rs		0	0	0	0	1	1	Rn		Rt	
	S																					

Encoding for the RCWSCASP variant

Applies when `(A == 0 && R == 0)`

```
RCWSCASP  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWSCASPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSCASPA  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWSCASPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSCASPAL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWSCASPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSCASPL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
if Rs<0> == '1' then EndOfDecode(Decode_UNDEF);
if Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean soft = TRUE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<X(s+1)>

Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>

Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<X(t+1)>

Is the 64-bit name of the second general-purpose register to be conditionally stored.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(128) newdata;
bits(128) compdata;
bits(128) readdata;
bits(4) nzcv;

constant bits(64) s1 = X[s, 64];
constant bits(64) s2 = X[s+1, 64];
constant bits(64) t1 = X[t, 64];
constant bits(64) t2 = X[t+1, 64];

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, soft, acquire, release,
                                                     tagchecked, t, t+1, s, s+1);

compdata = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newdata  = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
PSTATE.<N,Z,C,V> = nzcv;
if BigEndian(accdesc.acctype) then
    X[s, 64]   = readdata<127:64>;
    X[s+1, 64] = readdata<63:0>;
else
    X[s, 64]   = readdata<63:0>;
    X[s+1, 64] = readdata<127:64>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.327 RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL

Read check write software atomic bit clear on doubleword in memory

This instruction atomically loads a 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWSCLRA` and `RCWSCLRAL` load from memory with acquire semantics.
  - `RCWSCLRL` and `RCWSCLRAL` store to memory with release semantics.
  - `RCWSCLR` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

Encoding for the RCWSCLR variant

Applies when `(A == 0 && R == 0)`

```
RCWSCLR  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCLRA variant

Applies when `(A == 1 && R == 0)`

```
RCWSCLRA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCLRAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSCLRAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCLRL variant

Applies when `(A == 0 && R == 1)`

```
RCWSCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean soft = TRUE;

constant boolean acquire = A == '1' && t != 31;
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;

bits(64) address;
constant bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_BIC, soft, acquire, release,
                                                     tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.328 RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL

Read check write software atomic bit clear on quadword in memory

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSCLRPA` and `RCWSCLRPAL` load from memory with acquire semantics.
  - `RCWSCLRPL` and `RCWSCLRPAL` store to memory with release semantics.
  - `RCWSCLRP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

Encoding for the RCWSCLRP variant

Applies when `(A == 0 && R == 0)`

```
RCWSCLRP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSCLRPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSCLRPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSCLRPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSCLRPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSCLRPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSCLRPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean soft = TRUE;

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_BIC, soft, acquire, release,
                                                     tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.329 RCWSET, RCWSETA, RCWSETAL, RCWSETL

Read check write atomic bit set on doubleword in memory

This instruction atomically loads a 64-bit doubleword from memory, performs a bitwise OR with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWSETA` and `RCWSETAL` load from memory with acquire semantics.
  - `RCWSETL` and `RCWSETAL` store to memory with release semantics.
  - `RCWSET` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

Encoding for the RCWSET variant

Applies when `(A == 0 && R == 0)`

```
RCWSET  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSETA variant

Applies when `(A == 1 && R == 0)`

```
RCWSETA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSETAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSETAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSETL variant

Applies when `(A == 0 && R == 1)`

```
RCWSETL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean soft = FALSE;

constant boolean acquire = A == '1' && t != 31;
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;

bits(64) address;
constant bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, soft, acquire, release,
                                                     tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.330 RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL

Read check write atomic bit set on quadword in memory

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSETPA` and `RCWSETPAL` load from memory with acquire semantics.
  - `RCWSETPL` and `RCWSETPAL` store to memory with release semantics.
  - `RCWSETP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

Encoding for the RCWSETP variant

Applies when `(A == 0 && R == 0)`

```
RCWSETP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSETPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSETPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSETPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSETPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSETPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSETPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean soft = FALSE;

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, soft, acquire, release,
                                                     tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.331 RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL

Read check write software atomic bit set on doubleword in memory

This instruction atomically loads a 64-bit doubleword from memory, performs a bitwise OR with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWSSETA` and `RCWSSETAL` load from memory with acquire semantics.
  - `RCWSSETL` and `RCWSSETAL` store to memory with release semantics.
  - `RCWSSET` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

Encoding for the RCWSSET variant

Applies when `(A == 0 && R == 0)`

```
RCWSSET  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSETA variant

Applies when `(A == 1 && R == 0)`

```
RCWSSETA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSETAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSSETAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSETL variant

Applies when `(A == 0 && R == 1)`

```
RCWSSETL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean soft = TRUE;

constant boolean acquire = A == '1' && t != 31;
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;

bits(64) address;
constant bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, soft, acquire, release,
                                                     tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.332 RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL

Read check write software atomic bit set on quadword in memory

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSSETPA` and `RCWSSETPAL` load from memory with acquire semantics.
  - `RCWSSETPL` and `RCWSSETPAL` store to memory with release semantics.
  - `RCWSSETP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

Encoding for the RCWSSETP variant

Applies when `(A == 0 && R == 0)`

```
RCWSSETP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSETPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSSETPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSETPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSSETPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSETPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSSETPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean soft = TRUE;

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, soft, acquire, release,
                                                     tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.333 RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL

Read check write software swap doubleword in memory

This instruction atomically loads a 64-bit doubleword from a memory location, and conditionally stores the value held in a register back to the same memory location. Storing back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWSSWPA` and `RCWSSWPAL` load from memory with acquire semantics.
  - `RCWSSWPL` and `RCWSSWPAL` store to memory with release semantics.
  - `RCWSSWP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

Encoding for the RCWSSWP variant

Applies when `(A == 0 && R == 0)`

```
RCWSSWP  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSWPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSSWPA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSWPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSSWPAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSWPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSSWPL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean soft = TRUE;

constant boolean acquire = A == '1' && t != 31;
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
constant bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, soft, acquire, release,
                                                     tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;   // Return the old value when t!=31
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.334 RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL

Read check write software swap quadword in memory

This instruction atomically loads a 128-bit quadword from a memory location, and conditionally stores the value held in a pair of registers back to the same memory location. Storing back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSSWPPA` and `RCWSSWPPAL` load from memory with acquire semantics.
  - `RCWSSWPPL` and `RCWSSWPPAL` store to memory with release semantics.
  - `RCWSSWPP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

Encoding for the RCWSSWPP variant

Applies when `(A == 0 && R == 0)`

```
RCWSSWPP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSWPPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSSWPPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSWPPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSSWPPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSWPPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSSWPPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean soft = TRUE;

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, soft, acquire, release,
                                                     tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) compdata = bits(128) UNKNOWN; // Irrelevant when not executing CAS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64]  = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64]  = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.335 RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL

Read check write swap doubleword in memory

This instruction atomically loads a 64-bit doubleword from a memory location, and conditionally stores the value held in a register back to the same memory location. Storing back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWSWPA` and `RCWSWPAL` load from memory with acquire semantics.
  - `RCWSWPL` and `RCWSWPAL` store to memory with release semantics.
  - `RCWSWP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

Encoding for the RCWSWP variant

Applies when `(A == 0 && R == 0)`

```
RCWSWP  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSWPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSWPA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSWPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSWPAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSWPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSWPL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean soft = FALSE;

constant boolean acquire = A == '1' && t != 31;
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
constant bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, soft, acquire, release,
                                                     tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;   // Return the old value when t!=31
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.336 RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL

Read check write swap quadword in memory

This instruction atomically loads a 128-bit quadword from a memory location, and conditionally stores the value held in a pair of registers back to the same memory location. Storing back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSWPPA` and `RCWSWPPAL` load from memory with acquire semantics.
  - `RCWSWPPL` and `RCWSWPPAL` store to memory with release semantics.
  - `RCWSWPP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

31	30	29					24	23	22	21	20	16	15	14			12	11	10	9	5	4	0
0	0	0	1	1	0	0	1	A	R	1	Rt2		1	0	1	0		0	0	Rn		Rt	
	S												o3		opc								

Encoding for the RCWSWPP variant

Applies when `(A == 0 && R == 0)`

```
RCWSWPP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSWPPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSWPPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSWPPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSWPPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSWPPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSWPPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean soft = FALSE;

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

constant AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, soft, acquire, release,
                                                     tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) compdata = bits(128) UNKNOWN; // Irrelevant when not executing CAS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64]  = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64]  = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.337 RET

Return from subroutine

This instruction branches unconditionally to an address in a register. This instruction provides a hint that this is a subroutine return.

31–29	28–26	25	24	23	22–21	20–16	15–12	11	10	9–5	4–0
1 1 0	1 0 1	1	0	0	1 0	1 1 1 1 1	0 0 0 0	0	0	Rn	0 0 0 0 0
			Z		op	op2		A	M		Rm

Encoding

```
RET  {<Xn>}
```

Decode for this encoding

```
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field. Defaults to X30 if absent.

Operation

```
bits(64) target = X[n, 64];
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    target = LoadCheckGCSRecord(target, GCSInstType_PRET);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_RET, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.338 RETAA, RETAB

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, and branches to the authenticated address.

Key A is used for `RETAA`. Key B is used for `RETAB`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to LR.

This instruction provides a hint that this is a subroutine return.

If FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `RETAA` and `RETAB` include a second modifier that is in X16.

Integer

(FEAT_PAuth)

31–25	24	23	22–21	20–16	15–12	11	10	9–5	4–0
1 1 0 1 0 1 1	0	0	1 0	1 1 1 1 1	0 0 0 0	1	M	1 1 1 1 1	1 1 1 1 1
	Z		op	op2		A		Rn	Rm

Encoding for the RETAA variant

Applies when `(M == 0)`

```
RETAA
```

Encoding for the RETAB variant

Applies when `(M == 1)`

```
RETAB
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

constant boolean use_key_a = (M == '0');
constant boolean auth_then_branch = TRUE;
```

Operation

```
GCSInstruction inst_type;
bits(64) target = X[30, 64];

constant bits(64) modifier = SP[64];
bits(64) modifier2;
boolean use_modifier2 = FALSE;
if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
    modifier2 = X[16, 64];
    use_modifier2 = TRUE;

if use_key_a then
    if use_modifier2 && IsFeatureImplemented(FEAT_PAuth_LR) then
        target = AuthIA2(target, modifier, modifier2, auth_then_branch);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        target = AuthIA(target, modifier, auth_then_branch);
else
    if use_modifier2 && IsFeatureImplemented(FEAT_PAuth_LR) then
        target = AuthIB2(target, modifier, modifier2, auth_then_branch);
    else
        target = AuthIB(target, modifier, auth_then_branch);

if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    inst_type = if use_key_a then GCSInstType_PRETAA else GCSInstType_PRETAB;
    target = LoadCheckGCSRecord(target, inst_type);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_RET, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.339 RETAASPPC, RETABSPPC

Return from subroutine, with enhanced pointer authentication using an immediate offset

This instruction authenticates the address that is held in LR, using SP as the first modifier, the specified immediate subtracted from PC as the second modifier, and the specified key, and branches to the authenticated address.

Key A is used for `RETAASPPC`. Key B is used for `RETABSPPC`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to LR.

This instruction provides a hint that this is a subroutine return.

Integer

(FEAT_PAuth_LR)

31							24	23		21	20						5	4				0
0	1	0	1	0	1	0	1	0	0	x	imm16							1	1	1	1	1
								opc												op2		

Encoding for the RETAASPPC variant

Applies when `(opc == 000)`

```
RETAASPPC  <label>
```

Encoding for the RETABSPPC variant

Applies when `(opc == 001)`

```
RETABSPPC  <label>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);

constant boolean use_key_a = opc<0> == '0';
constant bits(64) offset = ZeroExtend(imm16:'00', 64);
constant boolean auth_then_branch = TRUE;
```

Assembler Symbols

<label>

Is the program label whose address is to be calculated. Its negative offset from the address of this instruction, a multiple of 4 in the range -262140 to 0, is encoded as an unsigned value in the “imm16” field as <label>/4.

Operation

```
GCSInstruction inst_type;
bits(64) target = X[30, 64];

constant bits(64) modifier = SP[64];
constant bits(64) modifier2 = PC64 - offset;

if use_key_a then
    target = AuthIA2(target, modifier, modifier2, auth_then_branch);
else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    target = AuthIB2(target, modifier, modifier2, auth_then_branch);

if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    inst_type = if use_key_a then GCSInstType_PRETAA else GCSInstType_PRETAB;
    target = LoadCheckGCSRecord(target, inst_type);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_RET, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.340 RETAASPPCR, RETABSPPCR

Return from subroutine, with enhanced pointer authentication using a register

This instruction authenticates the address that is held in LR, using SP as the first modifier, the value in the specified register as the second modifier, and the specified key, and branches to the authenticated address.

Key A is used for `RETAASPPCR`. Key B is used for `RETABSPPCR`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to LR.

This instruction provides a hint that this is a subroutine return.

Integer

(FEAT_PAuth_LR)

31–25	24–21	20–16	15–11	10	9–5	4–0
1 1 0 1 0 1 1	0 0 1 0	1 1 1 1 1	0 0 0 0 1	M	1 1 1 1 1	!=11111
	opc	op2			Rn	Rm

Encoding for the RETAASPPCR variant

Applies when `(M == 0)`

```
RETAASPPCR  <Xm>
```

Encoding for the RETABSPPCR variant

Applies when `(M == 1)`

```
RETABSPPCR  <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF);
constant integer m = UInt(Rm);
constant boolean use_key_a = M == '0';
constant boolean auth_then_branch = TRUE;
```

Assembler Symbols

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

```
GCSInstruction inst_type;
bits(64) target = X[30, 64];

constant bits(64) modifier = SP[64];
constant bits(64) modifier2 = X[m, 64];

if use_key_a then
    target = AuthIA2(target, modifier, modifier2, auth_then_branch);
else
    target = AuthIB2(target, modifier, modifier2, auth_then_branch);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    inst_type = if use_key_a then GCSInstType_PRETAA else GCSInstType_PRETAB;
    target = LoadCheckGCSRecord(target, inst_type);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_RET, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.341 REV

Reverse bytes

This instruction reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.

31	30	29	28							21	20					16	15				12	11	10	9				5	4				0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0		0	0	0	0		1	x	Rn					Rd				
		S									opcode2											opc											

Encoding for the 32-bit variant

Applies when `(sf == 0 && opc == 10)`

```
REV  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && opc == 11)`

```
REV  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    constant bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.342 REV16

Reverse bytes in 16-bit halfwords

This instruction reverses the byte order in each 16-bit halfword of a register.

31	30	29	28							21	20					16	15				12	11	10	9				5	4				0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0		0	0	0	0		0	1	Rn					Rd				
		S													opcode2								opc										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
REV16  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
REV16  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    constant bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.343 REV32

Reverse bytes in 32-bit words

This instruction reverses the byte order in each 32-bit word of a register.

Encoding

```
REV32  <Xd>, <Xn>
```

Decode for this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    constant bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.344 REV64

Reverse bytes

This instruction 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.

This is a pseudo-instruction of REV. This means:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding

```
REV64  <Xd>, <Xn>
```

is equivalent to

```
REV  <Xd>, <Xn>
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational Information

The description of REV gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.345 RMIF

Rotate, mask insert flags

This instruction performs a rotation right of a value held in a general-purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer

(FEAT_FlagM)

Encoding

```
RMIF  <Xn>, #<shift>, #<mask>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF);
constant integer imm = UInt(imm6);
constant bits(4) flagmask = mask;
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

<shift>

Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<mask>

Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the “mask” field.

Operation

```
constant bits(64) reg = X[n, 64];
constant bits(4) flags = (reg:reg)<imm+3:imm>;
if flagmask<3> == '1' then PSTATE.N = flags<3>;
if flagmask<2> == '1' then PSTATE.Z = flags<2>;
if flagmask<1> == '1' then PSTATE.C = flags<1>;
if flagmask<0> == '1' then PSTATE.V = flags<0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.346 ROR (immediate)

Rotate right (immediate)

This instruction 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.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21	20 … 16	15 … 10	9 … 5	4 … 0
sf	0 0	1 0 0 1 1 1	N	0	Rm	imms	Rn	Rd
	op21			o0				

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms == 0xxxxx)`

```
ROR  <Wd>, <Ws>, #<shift>
```

is equivalent to

```
EXTR  <Wd>, <Ws>, <Ws>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
ROR  <Xd>, <Xs>, #<shift>
```

is equivalent to

```
EXTR  <Xd>, <Xs>, <Xs>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Ws>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

<shift>

For the “32-bit” variant: is the amount by which to rotate, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the amount by which to rotate, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xs>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Operational Information

The description of EXTR gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.347 ROR (register)

Rotate right (register)

This instruction 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 value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ROR  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
RORV  <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ROR  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
RORV  <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational Information

The description of RORV gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.348 RORV

Rotate right variable

This instruction 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 value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register).

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0		1	1	Rn					Rd				
		S																				op2										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
RORV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
RORV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
ROR (register)	Unconditionally

Operation

```
constant bits(datasize) operand2 = X[m, datasize];

X[d, datasize] = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.349 RPRFM

Range prefetch memory

This instruction signals the memory system that data memory accesses from a specified range of addresses are likely to occur in the near future. The instruction may also signal the memory system about the likelihood of data reuse of the specified range of addresses. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as prefetching locations within the specified address ranges into one or more caches. The memory system may also exploit the data reuse hints to decide whether to retain the data in other caches upon eviction from the innermost caches or to discard it.

The effect of an `RPRFM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

An `RPRFM` instruction specifies the type of accesses and range of addresses using the following parameters:

- ‘Access type’, encoded in `Rt<0>`, specifies:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- ‘Policy’, encoded in `Rt<2:1>`, specifies:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once. If a streaming prefetch is specified, then the ‘ReuseDistance’ parameter is ignored.
- ‘BaseAddress’, in the 64-bit base register, holds the initial block address for the accesses.
- ‘ReuseDistance’, in the metadata register bits[63:60], indicates the maximum number of bytes to be accessed by this PE before executing the next `RPRFM` instruction that specifies the same range. This includes the total number of bytes inside and outside of the range that will be accessed by the same PE. This parameter can be used to influence cache eviction and replacement policies, in order to retain the data in the most optimal levels of the memory hierarchy after each access. If software cannot easily determine the amount of other memory that will be accessed, these bits can be set to zero to indicate that ‘ReuseDistance’ is not known. Otherwise, these four bits encode decreasing powers of two in the range 512MiB (`0b0001`) to 32KiB (`0b1111`).
- ‘Stride’, in the metadata register bits[59:38], is a signed, two’s complement integer encoding of the number of bytes to advance the block address after ‘Length’ bytes have been accessed, in the range -2MiB to +2MiB-1B. A negative value indicates that the block address is advanced in a descending direction.
- ‘Count’, in the metadata register bits[37:22], is an unsigned integer encoding of the number of blocks of data to be accessed minus 1, representing the range 1 to 65536 blocks. If ‘Count’ is 0, then the ‘Stride’ parameter is ignored and only a single block of contiguous bytes from ‘BaseAddress’ to (‘BaseAddress’ + ‘Length’ - 1) is described.
- ‘Length’, in the metadata register bits[21:0], is a signed, two’s complement integer encoding of the number of contiguous bytes to be accessed starting from the current block address, without changing the block address, in the range -2MiB to +2MiB-1B. A negative value indicates that the bytes are accessed in a descending direction.

Note

Software is expected to honor the parameters it provides to the `RPRFM` instruction, and the same PE should access all locations in the range, in the direction specified by the sign of the ‘Length’ and ‘Stride’ parameters. A range prefetch is considered active on a PE until all locations in the range have been accessed by the PE. A range prefetch might also be inactivated by the PE prior to completion, for example due to a software context switch or lack of hardware resources. Software should not specify overlapping addresses in multiple active ranges. If a range is expected to be accessed by both load and store instructions (read-modify-write), then a single range with an ‘Access type’ parameter of `PST` (prefetch for store) should be specified.

Integer

(FEAT_RPRFM)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	Rm	x	1	x	S	1	0	Rn	1	1	x	x	x
size					VR			opc				option							Rt				

Encoding

```
RPRFM  (<rprfop>|#<imm6>), <Xm>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RPRFM) then EndOfDecode(Decode_NOP);
constant bits(6) operation = option<2>:option<0>:S:Rt<2:0>;
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<rprfop>

Is the range prefetch operation, encoded in “option:S:Rt”:

option	S	Rt	<rprfop>
010	0	11000	PLDKEEP
010	0	11001	PSTKEEP
010	0	11100	PLDSTRM
010	0	11101	PSTSTRM

For other encodings, use <imm6>.

<imm6>

Is the range prefetch operation encoding as an immediate, in the range 0 to 63, encoded in “option<2>:option<0>:S:Rt<2:0>”. This syntax is only for encodings that are not representable using `<rprfop>`.

<Xm>

Is the 64-bit name of the general-purpose register that holds an encoding of the metadata, encoded in the “Rm” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
constant bits(64) address  = if n == 31 then SP[64] else X[n, 64];
constant bits(64) metadata = X[m, 64];
constant integer stride = SInt(metadata<59:38>);
constant integer count  = UInt(metadata<37:22>) + 1;
constant integer length = SInt(metadata<21:0>);
integer reuse;

if metadata<63:60> == '0000' then
    reuse = -1; // Not known
else
    reuse = 32768 << (15 - UInt(metadata<63:60>));

Hint_RangePrefetch(address, length, stride, count, reuse, operation);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.350 SB

Speculation barrier

This instruction is a barrier that controls speculation. For more information and details of the semantics, see Speculation Barrier (SB).

System

(FEAT_SB)

31–29	28–26	25–12	11–8	7	6–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	(0) (0) (0) (0)	1	1 1	1 1 1 1 1
			CRm		opc	Rt

Encoding

```
SB
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SB) then EndOfDecode(Decode_UNDEF);
```

Operation

```
SpeculationBarrier();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.351 SBC

Subtract with carry

This instruction subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC.

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	1	0	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0		Rn					Rd				
	op	S																														

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SBC  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SBC  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
NGC	Rn == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = NOT(X[m, datasize]);
bits(datasize) result;

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

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.352 SBCS

Subtract with carry, setting flags

This instruction subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS.

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	1	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0		Rn					Rd				
	op																															
		S																														

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SBCS  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SBCS  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
NGCS	Rn == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = NOT(X[m, datasize]);
bits(datasize) result;
bits(4) nzcv;

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

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.353 SBFIZ

Signed bitfield insert in zeros

This instruction copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SBFIZ  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #(-<lsb>  MOD  32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SBFIZ  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #(-<lsb>  MOD  64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.354 SBFM

Signed bitfield move

This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If `<imms>` is greater than or equal to `<immr>`, this copies a bitfield of (`<imms>`-`<immr>`+1) bits starting from bit position `<immr>` in the source register to the least significant bits of the destination register.

If `<imms>` is less than `<immr>`, this copies a bitfield of (`<imms>`+1) bits from the least significant bits of the source register to bit position (regsize-`<immr>`) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd
	opc						

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SBFM  <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SBFM  <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF);
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer s = UInt(imms);
constant integer r = UInt(immr);

bits(datasize) wmask;
bits(datasize) tmask;
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<immr>

For the “32-bit” variant: is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.

<imms>

For the “32-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Alias	Of variant	Is preferred when
ASR (immediate)	32-bit	imms == '011111'
ASR (immediate)	64-bit	imms == '111111'
SBFIZ		UInt(imms) < UInt(immr)
SBFX		BFXPreferred(sf, opc<1>, imms, immr)
SXTB		immr == '000000' && imms == '000111'
SXTH		immr == '000000' && imms == '001111'
SXTW		immr == '000000' && imms == '011111'

Operation

```
constant bits(datasize) src = X[n, datasize];

// Perform bitfield move on low bits
constant bits(datasize) bot = ROR(src, r) AND wmask;

constant bits(datasize) top = Replicate(src<s>, datasize);

// Combine extension bits and result bits
X[d, datasize] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.355 SBFX

Signed bitfield extract

This instruction copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SBFX  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc[1], imms, immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SBFX  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc[1], imms, immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.356 SDIV

Signed divide

This instruction divides the first signed source register value by the second signed source register value, and writes the result to the destination register. Dividing by zero writes the value zero to the destination register. The condition flags are not affected.

31	30	29	28	21	20	16	15	11	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 0 0 0 1		1	Rn		Rd	
		S							o1				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SDIV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SDIV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant integer dividend = SInt(operand1);
constant integer divisor  = SInt(operand2);
integer result;
if divisor == 0 then
    result = 0;
elsif (dividend < 0) == (divisor < 0) then
    result = Abs(dividend) DIV Abs(divisor);    // same signs - positive result
else
    result = -(Abs(dividend) DIV Abs(divisor)); // different signs - negative result
X[d, datasize] = result<datasize-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.357 SETF8, SETF16

Evaluation of 8-bit or 16-bit flag values

This instruction sets the PSTATE.NZV flags based on the value in the specified general-purpose register. `SETF8` treats the value as an 8-bit value. `SETF16` treats the value as a 16-bit value.

The PSTATE.C flag is not affected by these instructions.

Integer

(FEAT_FlagM)

Encoding for the SETF8 variant

Applies when `(sz == 0)`

```
SETF8  <Wn>
```

Encoding for the SETF16 variant

Applies when `(sz == 1)`

```
SETF16  <Wn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF);
constant integer size = 8 << UInt(sz);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(32) reg = X[n, 32];
PSTATE.N = reg<size-1>;
PSTATE.Z = if (reg<size-1:0> == Zeros(size)) then '1' else '0';
PSTATE.V = reg<size> EOR reg<size-1>;
//PSTATE.C unchanged;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.358 SETGP, SETGM, SETGE

Memory set with tag setting

These instructions set a requested number of bytes in memory to the value in the least significant byte of the source data register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register that holds the first address to be set. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGP, then SETGM, and then SETGE.

SETGP performs some preconditioning of the arguments suitable for using the SETGM instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETGM sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETGE sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETGP:

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.

On completion of SETGP, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETGP, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETGM, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETGM, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETGE, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On completion of the instruction, Xn holds 0.

For SETGE, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Integer

(FEAT_MOPS && FEAT_MTE)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	1 1	0	Rs	x x 0 0	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0000)`

```
SETGP  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0100)`

```
SETGM  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1000)`

```
SETGE  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then
    EndOfDecode(Decode_UNDEF);

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
constant bits(2) options = op2<1:0>;
constant boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise EndOfDecode(Decode_UNDEF);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the “Rd” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero on completion of the instruction, encoded in the “Rn” field.

<Xs>

For the “Main” and “Prologue” variants: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the “Rs” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the “Rs” field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset.n, memset.d, memset.s);

constant bits(8) data = X[memset.s, 8];
MOPSBlockSize B = 0;

memset.is_setg = TRUE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETGOptionA();

constant boolean privileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                               else PSTATE.EL != EL0);

constant AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSSETGSize then
        memset.setsize = ArchMaxMOPSSETGSize;

    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
            !IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        constant FaultRecord fault = AlignmentFault(accdesc, memset.toaddress);
        AArch64.Abort(fault);

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    bits(64) fault_address;
    if memset.implements_option_a then
        fault_address = memset.toaddress + memset.setsize;
    else
        fault_address = memset.toaddress;

    if (memset.setsize != 0 && (memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
          !IsAligned(memset.toaddress, TAG_GRANULE)) then
        constant FaultRecord fault = AlignmentFault(accdesc, fault_address);
        AArch64.Abort(fault);
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
           !IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        constant FaultRecord fault = AlignmentFault(accdesc, fault_address);
        AArch64.Abort(fault);

integer tags_set;
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
AddressDescriptor tagmemaddrdesc;
PhysMemRetStatus  tagmemstatus;
boolean fault    = FALSE;
boolean tagfault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault && !tagfault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= -1 * memset.stagesetsize && B<3:0> == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                  data, B, accdesc);
        constant bits(4) tag = AArch64.AllocationTagFromAddress(memset.toaddress + memset.setsize);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress + memset.setsize,
                                                              tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault && !tagfault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= memset.stagesetsize && B<3:0> == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);
        constant bits(4) tag = AArch64.AllocationTagFromAddress(memset.toaddress);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress, tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.toaddress    = memset.toaddress    + B;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault || tagfault, tags_set * TAG_GRANULE);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
elsif tagfault then
    if IsFault(tagmemaddrdesc) then
        AArch64.Abort(tagmemaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(tagmemstatus, iswrite, tagmemaddrdesc, 1, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.359 SETGPN, SETGMN, SETGEN

Memory set with tag setting, non-temporal

These instructions set a requested number of bytes in memory to the value in the least significant byte of the source data register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register that holds the first address to be set. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPN, then SETGMN, and then SETGEN.

SETGPN performs some preconditioning of the arguments suitable for using the SETGMN instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETGMN sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETGEN sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETGPN:

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.

On completion of SETGPN, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETGPN, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETGMN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETGMN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETGEN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

For SETGEN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Integer

(FEAT_MOPS && FEAT_MTE)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	1 1	0	Rs	x x 1 0	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0010)`

```
SETGPN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0110)`

```
SETGMN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1010)`

```
SETGEN  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then
    EndOfDecode(Decode_UNDEF);

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
constant bits(2) options = op2<1:0>;
constant boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise EndOfDecode(Decode_UNDEF);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the “Rd” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero on completion of the instruction, encoded in the “Rn” field.

<Xs>

For the “Main” and “Prologue” variants: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the “Rs” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the “Rs” field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset.n, memset.d, memset.s);

constant bits(8) data = X[memset.s, 8];
MOPSBlockSize B = 0;

memset.is_setg = TRUE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETGOptionA();

constant boolean privileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                               else PSTATE.EL != EL0);

constant AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSSETGSize then
        memset.setsize = ArchMaxMOPSSETGSize;

    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
            !IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        constant FaultRecord fault = AlignmentFault(accdesc, memset.toaddress);
        AArch64.Abort(fault);

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memset.nzcv = '0010';

        memset.stagesetsize = MemSetStageSize(memset);

        if memset.stage != MOPSStage_Prologue then
            CheckMemSetParams(memset, options);

            bits(64) fault_address;
            if memset.implements_option_a then
                fault_address = memset.toaddress + memset.setsize;
            else
                fault_address = memset.toaddress;

            if (memset.setsize != 0 && (memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
                  !IsAligned(memset.toaddress, TAG_GRANULE)) then
                constant FaultRecord fault = AlignmentFault(accdesc, fault_address);
                AArch64.Abort(fault);
            if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
                   !IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
                constant FaultRecord fault = AlignmentFault(accdesc, fault_address);
                AArch64.Abort(fault);

        integer tags_set;
        AddressDescriptor memaddrdesc;
        PhysMemRetStatus  memstatus;
        AddressDescriptor tagmemaddrdesc;
        PhysMemRetStatus  tagmemstatus;
        boolean fault    = FALSE;
        boolean tagfault = FALSE;

        if memset.implements_option_a then
            while memset.stagesetsize < 0 && !fault && !tagfault do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = SETSizeChoice(memset, TAG_GRANULE);
                assert B <= -1 * memset.stagesetsize && B<3:0> == '0000';

                (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                          data, B, accdesc);
                constant bits(4) tag = AArch64.AllocationTagFromAddress(memset.toaddress + memset.setsize);
                (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress + memset.setsize,
                                                                      tag, B, accdesc);

                fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
                tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

                if !fault && !tagfault then
                    memset.setsize      = memset.setsize      + B;
                    memset.stagesetsize = memset.stagesetsize + B;

        else
            while memset.stagesetsize > 0 && !fault && !tagfault do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = SETSizeChoice(memset, TAG_GRANULE);
                assert B <= memset.stagesetsize && B<3:0> == '0000';

                (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);
                constant bits(4) tag = AArch64.AllocationTagFromAddress(memset.toaddress);
                (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress, tag, B, accdesc);

                fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
                tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

                if !fault && !tagfault then
                    memset.toaddress    = memset.toaddress    + B;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault || tagfault, tags_set * TAG_GRANULE);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
elsif tagfault then
    if IsFault(tagmemaddrdesc) then
        AArch64.Abort(tagmemaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(tagmemstatus, iswrite, tagmemaddrdesc, 1, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.360 SETGPT, SETGMT, SETGET

Memory set with tag setting, unprivileged

These instructions set a requested number of bytes in memory to the value in the least significant byte of the source data register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register that holds the first address to be set. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPT, then SETGMT, and then SETGET.

SETGPT performs some preconditioning of the arguments suitable for using the SETGMT instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETGMT sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETGET sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETGPT:

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.

On completion of SETGPT, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETGPT, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETGMT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETGMT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETGET, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

For SETGET, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS && FEAT_MTE)

Encoding for the Prologue variant

Applies when `(op2 == 0001)`

```
SETGPT  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0101)`

```
SETGMT  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1001)`

```
SETGET  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then
    EndOfDecode(Decode_UNDEF);

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
constant bits(2) options = op2<1:0>;
constant boolean nontemporal = options<1> == '1';

case op2<3:2> of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise EndOfDecode(Decode_UNDEF);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero on completion of the instruction, encoded in the "Rn" field.

<Xs>

For the "Main" and "Prologue" variants: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset.n, memset.d, memset.s);

constant bits(8) data = X[memset.s, 8];
MOPSBlockSize B = 0;

memset.is_setg = TRUE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETGOptionA();

constant boolean privileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                               else PSTATE.EL != EL0);

constant AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSSETGSize then
        memset.setsize = ArchMaxMOPSSETGSize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
                !IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
            constant FaultRecord fault = AlignmentFault(accdesc, memset.toaddress);
            AArch64.Abort(fault);

        if memset.implements_option_a then
            memset.nzcv = '0000';
            memset.toaddress = memset.toaddress + memset.setsize;
            memset.setsize   = 0 - memset.setsize;
        else
            memset.nzcv = '0010';

    memset.stagesetsize = MemSetStageSize(memset);

    if memset.stage != MOPSStage_Prologue then
        CheckMemSetParams(memset, options);

        bits(64) fault_address;
        if memset.implements_option_a then
            fault_address = memset.toaddress + memset.setsize;
        else
            fault_address = memset.toaddress;

        if (memset.setsize != 0 && (memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
              !IsAligned(memset.toaddress, TAG_GRANULE)) then
            constant FaultRecord fault = AlignmentFault(accdesc, fault_address);
            AArch64.Abort(fault);
        if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
               !IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
            constant FaultRecord fault = AlignmentFault(accdesc, fault_address);
            AArch64.Abort(fault);

    integer tags_set;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    AddressDescriptor tagmemaddrdesc;
    PhysMemRetStatus  tagmemstatus;
    boolean fault    = FALSE;
    boolean tagfault = FALSE;

    if memset.implements_option_a then
        while memset.stagesetsize < 0 && !fault && !tagfault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = SETSizeChoice(memset, TAG_GRANULE);
            assert B <= -1 * memset.stagesetsize && B<3:0> == '0000';

            (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                      data, B, accdesc);
            constant bits(4) tag = AArch64.AllocationTagFromAddress(memset.toaddress + memset.setsize);
            (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress + memset.setsize,
                                                                  tag, B, accdesc);

            fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
            tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

            if !fault && !tagfault then
                memset.setsize      = memset.setsize      + B;
                memset.stagesetsize = memset.stagesetsize + B;

    else
        while memset.stagesetsize > 0 && !fault && !tagfault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = SETSizeChoice(memset, TAG_GRANULE);
            assert B <= memset.stagesetsize && B<3:0> == '0000';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);
        constant bits(4) tag = AArch64.AllocationTagFromAddress(memset.toaddress);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress, tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault || tagfault, tags_set * TAG_GRANULE);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
elsif tagfault then
    if IsFault(tagmemaddrdesc) then
        AArch64.Abort(tagmemaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(tagmemstatus, iswrite, tagmemaddrdesc, 1, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.361 SETGPTN, SETGMTN, SETGETN

Memory set with tag setting, unprivileged and non-temporal

These instructions set a requested number of bytes in memory to the value in the least significant byte of the source data register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register that holds the first address to be set. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPTN, then SETGMTN, and then SETGETN.

SETGPTN performs some preconditioning of the arguments suitable for using the SETGMTN instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETGMTN sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETGETN sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETGPTN:

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.

On completion of SETGPTN, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETGPTN, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETGMTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETGMTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETGETN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

For SETGETN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS && FEAT_MTE)

Encoding for the Prologue variant

Applies when `(op2 == 0011)`

```
SETGPTN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0111)`

```
SETGMTN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1011)`

```
SETGETN  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then
    EndOfDecode(Decode_UNDEF);

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
constant bits(2) options = op2<1:0>;
constant boolean nontemporal = options<1> == '1';

case op2<3:2> of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise EndOfDecode(Decode_UNDEF);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero on completion of the instruction, encoded in the "Rn" field.

<Xs>

For the "Main" and "Prologue" variants: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset.n, memset.d, memset.s);

constant bits(8) data = X[memset.s, 8];
MOPSBlockSize B = 0;

memset.is_setg = TRUE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETGOptionA();

constant boolean privileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                               else PSTATE.EL != EL0);

constant AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSSETGSize then
        memset.setsize = ArchMaxMOPSSETGSize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
                !IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
            constant FaultRecord fault = AlignmentFault(accdesc, memset.toaddress);
            AArch64.Abort(fault);

        if memset.implements_option_a then
            memset.nzcv = '0000';
            memset.toaddress = memset.toaddress + memset.setsize;
            memset.setsize   = 0 - memset.setsize;
        else
            memset.nzcv = '0010';

    memset.stagesetsize = MemSetStageSize(memset);

    if memset.stage != MOPSStage_Prologue then
        CheckMemSetParams(memset, options);

        bits(64) fault_address;
        if memset.implements_option_a then
            fault_address = memset.toaddress + memset.setsize;
        else
            fault_address = memset.toaddress;

        if (memset.setsize != 0 && (memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
              !IsAligned(memset.toaddress, TAG_GRANULE)) then
            constant FaultRecord fault = AlignmentFault(accdesc, fault_address);
            AArch64.Abort(fault);
        if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
               !IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
            constant FaultRecord fault = AlignmentFault(accdesc, fault_address);
            AArch64.Abort(fault);

    integer tags_set;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    AddressDescriptor tagmemaddrdesc;
    PhysMemRetStatus  tagmemstatus;
    boolean fault    = FALSE;
    boolean tagfault = FALSE;

    if memset.implements_option_a then
        while memset.stagesetsize < 0 && !fault && !tagfault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = SETSizeChoice(memset, TAG_GRANULE);
            assert B <= -1 * memset.stagesetsize && B<3:0> == '0000';

            (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                      data, B, accdesc);
            constant bits(4) tag = AArch64.AllocationTagFromAddress(memset.toaddress + memset.setsize);
            (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress + memset.setsize,
                                                                  tag, B, accdesc);

            fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
            tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

            if !fault && !tagfault then
                memset.setsize      = memset.setsize      + B;
                memset.stagesetsize = memset.stagesetsize + B;

    else
        while memset.stagesetsize > 0 && !fault && !tagfault do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = SETSizeChoice(memset, TAG_GRANULE);
            assert B <= memset.stagesetsize && B<3:0> == '0000';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);
        constant bits(4) tag = AArch64.AllocationTagFromAddress(memset.toaddress);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress, tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault || tagfault, tags_set * TAG_GRANULE);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
elsif tagfault then
    if IsFault(tagmemaddrdesc) then
        AArch64.Abort(tagmemaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(tagmemstatus, iswrite, tagmemaddrdesc, 1, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.362 SETP, SETM, SETE

Memory set

These instructions set a required number of bytes in memory to the value in the least significant byte of the source data register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETP, then SETM, and then SETE.

SETP performs some preconditioning of the arguments suitable for using the SETM instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETM sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETE sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETP:

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of SETP, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETP, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETM, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETM, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETE, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SETE, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	1 1	0	Rs	x x 0 0	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0000)`

```
SETP  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0100)`

```
SETM  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1000)`

```
SETE  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
constant bits(2) options = op2<1:0>;
constant boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise EndOfDecode(Decode_UNDEF);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the “Rd” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero on completion of the instruction, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register that holds the source data, encoded in the “Rs” field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset.n, memset.d, memset.s);

constant bits(8) data = X[memset.s, 8];
MOPSBlockSize B = 0;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETOptionA();

constant boolean privileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                               else PSTATE.EL != EL0);

constant AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSBlockSize then
        memset.setsize = ArchMaxMOPSBlockSize;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
integer memory_set;
boolean fault = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                            data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.363 SETPN, SETMN, SETEN

Memory set, non-temporal

These instructions set a required number of bytes in memory to the value in the least significant byte of the source data register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPN, then SETMN, and then SETEN.

SETPN performs some preconditioning of the arguments suitable for using the SETMN instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETMN sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETEN sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETPN:

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of SETPN, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETPN, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETMN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETMN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETEN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SETEN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	1 1	0	Rs	x x 1 0	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0010)`

```
SETPN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0110)`

```
SETMN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1010)`

```
SETEN  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
constant bits(2) options = op2<1:0>;
constant boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise EndOfDecode(Decode_UNDEF);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the “Rd” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero on completion of the instruction, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register that holds the source data, encoded in the “Rs” field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset.n, memset.d, memset.s);

constant bits(8) data = X[memset.s, 8];
MOPSBlockSize B = 0;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETOptionA();

constant boolean privileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                               else PSTATE.EL != EL0);

constant AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSBlockSize then
        memset.setsize = ArchMaxMOPSBlockSize;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
integer memory_set;
boolean fault = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                            data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.364 SETPT, SETMT, SETET

Memory set, unprivileged

These instructions set a required number of bytes in memory to the value in the least significant byte of the source data register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPT, then SETMT, and then SETET.

SETPT performs some preconditioning of the arguments suitable for using the SETMT instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETMT sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETET sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETPT:

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of SETPT, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETPT, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETMT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETMT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETET, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SETET, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	1 1	0	Rs	x x 0 1	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0001)`

```
SETPT  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0101)`

```
SETMT  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1001)`

```
SETET  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
constant bits(2) options = op2<1:0>;
constant boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise EndOfDecode(Decode_UNDEF);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero on completion of the instruction, encoded in the "Rn" field.

<Xs>

Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset.n, memset.d, memset.s);

constant bits(8) data = X[memset.s, 8];
MOPSBlockSize B = 0;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETOptionA();

constant boolean privileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                               else PSTATE.EL != EL0);

constant AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSBlockSize then
        memset.setsize = ArchMaxMOPSBlockSize;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                            data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.365 SETPTN, SETMTN, SETETN

Memory set, unprivileged and non-temporal

These instructions set a required number of bytes in memory to the value in the least significant byte of the source data register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPTN, then SETMTN, and then SETETN.

SETPTN performs some preconditioning of the arguments suitable for using the SETMTN instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETMTN sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETETN sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETPTN:

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of SETPTN, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETPTN, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETMTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETMTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETETN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SETETN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	1 1	0	Rs	x x 1 1	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0011)`

```
SETPTN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0111)`

```
SETMTN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1011)`

```
SETETN  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF);

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
constant bits(2) options = op2<1:0>;
constant boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise EndOfDecode(Decode_UNDEF);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero on completion of the instruction, encoded in the "Rn" field.

<Xs>

Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset.n, memset.d, memset.s);

constant bits(8) data = X[memset.s, 8];
MOPSBlockSize B = 0;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETOptionA();

constant boolean privileged = (if options<0> == '1' then AArch64.IsUnprivAccessPriv()
                               else PSTATE.EL != EL0);

constant AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSBlockSize then
        memset.setsize = ArchMaxMOPSBlockSize;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus  memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                            data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
    else
        constant boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.366 SEV

Send event

This instruction is a hint instruction that causes an event to be signaled to all PEs in the multiprocessor system. For more information, see Wait for Event mechanism and Send event.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0	1 0 0	1 1 1 1 1
			CRm	op2	

Encoding

```
SEV
```

Decode for this encoding

```
// Empty.
```

Operation

```
SendEvent();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.367 SEVL

Send event local

This instruction is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop that starts with a `WFE` instruction.

31	29	28	26	25	12	11	8	7	5	4	0
1 1 0		1 0 1		0 1 0 0 0 0 0 0 1 1 0 0 1 0		0 0 0 0 (CRm)		1 0 1 (op2)		1 1 1 1 1	

Encoding

```
SEVL
```

Decode for this encoding

```
// Empty.
```

Operation

```
SendEventLocal();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.368 SMADDL

Signed multiply-add long

This instruction multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	0 1	Rm	0	Ra	Rn	Rd
sf	op54				U			o0			

Encoding

```
SMADDL  <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
SMULL	Ra == '11111'

Operation

```
constant bits(32) operand1 = X[n, 32];
constant bits(32) operand2 = X[m, 32];
constant bits(64) operand3 = X[a, 64];

constant integer result = SInt(operand3) + (SInt(operand1) * SInt(operand2));

X[d, 64] = result<63:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.369 SMAX (immediate)

Signed maximum (immediate)

This instruction determines the signed maximum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28						22	21			18	17						10	9				5	4					0
sf	0	0	1	0	0	0	1	1	1	0	0	0	0	imm8							Rn					Rd					
op		S										opc																			

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SMAX  <Wd>, <Wn>, #<simm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SMAX  <Xd>, <Xn>, #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer imm = SInt(imm8);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<simm>

Is a signed immediate, in the range -128 to 127, encoded in the “imm8” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = Max(SInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.370 SMAX (register)

Signed maximum (register)

This instruction determines the signed maximum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	1	0	0	0		Rn					Rd				
		S																	opcode													

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SMAX  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SMAX  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant integer result = Max(SInt(operand1), SInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.371 SMC

Secure monitor call

This instruction causes an exception to EL3.

`SMC` is available only for software executing at EL1 or higher. It is UNDEFINED at 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 syndrome value `0x17`, that is taken to EL3.

If the value of HCR_EL2.TSC is 1 and EL2 is enabled in the current Security state, execution of an `SMC` instruction at EL1 generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.

Encoding

```
SMC  #<imm>
```

Decode for this encoding

```
constant bits(16) imm = imm16;
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;
AArch64.CheckForSMCUndefOrTrap(imm);
AArch64.CallSecureMonitor(imm);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.372 SMIN (immediate)

Signed minimum (immediate)

This instruction determines the signed minimum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28						22	21			18	17	10	9	5	4	0
sf	0	0	1	0	0	0	1	1	1	0	0	1	0	imm8		Rn		Rd	
op		S										opc							

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SMIN  <Wd>, <Wn>, #<simm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SMIN  <Xd>, <Xn>, #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer imm = SInt(imm8);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<simm>

Is a signed immediate, in the range -128 to 127, encoded in the “imm8” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = Min(SInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.373 SMIN (register)

Signed minimum (register)

This instruction determines the signed minimum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	1	0	1	0		Rn					Rd				
		S																	opcode													

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SMIN  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SMIN  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant integer result = Min(SInt(operand1), SInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.374 SMNEGL

Signed multiply-negate long

This instruction multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of SMSUBL. This means:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	0 1	Rm	1	1 1 1 1 1	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
SMNEGL  <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMSUBL  <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Operational Information

The description of SMSUBL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.375 SMSTART

Enables access to Streaming SVE mode and SME architectural state

This instruction enables access to Streaming SVE mode and SME architectural state.

SMSTART enters Streaming SVE mode, and enables the SME ZA storage.

SMSTART SM enters Streaming SVE mode, but does not enable the SME ZA storage.

SMSTART ZA enables the SME ZA storage, but does not cause an entry to Streaming SVE mode.

This is an alias of MSR (immediate). This means:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SME)

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	0 1 1	0 1 0 0	0 x x 1	0 1 1	1 1 1 1 1
			op1		CRm	op2	Rt

Encoding

```
SMSTART  {<option>}
```

is equivalent to

```
MSR  <pstatefield>,   #1
```

and is always the preferred disassembly.

Assembler Symbols

<option>

Is an optional mode, encoded in “CRm<2:1>”:

CRm<2:1>	<option>	Description
00	RESERVED	•
01	SM	Maps to <pstatefield> SVCRSM.
10	ZA	Maps to <pstatefield> SVCRZA.
11	[absent]	Maps to <pstatefield> SVCRSMZA.

Operation

The description of MSR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.376 SMSTOP

Disables access to Streaming SVE mode and SME architectural state

This instruction disables access to Streaming SVE mode and SME architectural state.

SMSTOP exits Streaming SVE mode, and disables the SME ZA storage.

SMSTOP SM exits Streaming SVE mode, but does not disable the SME ZA storage.

SMSTOP ZA disables the SME ZA storage, but does not cause an exit from Streaming SVE mode.

This is an alias of MSR (immediate). This means:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SME)

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	0 1 1	0 1 0 0	0 x x 0	0 1 1	1 1 1 1 1
			op1		CRm	op2	Rt

Encoding

```
SMSTOP  {<option>}
```

is equivalent to

```
MSR  <pstatefield>,   #0
```

and is always the preferred disassembly.

Assembler Symbols

<option>

Is an optional mode, encoded in “CRm<2:1>”:

CRm<2:1>	<option>	Description
00	RESERVED	•
01	SM	Maps to <pstatefield> SVCRSM.
10	ZA	Maps to <pstatefield> SVCRZA.
11	[absent]	Maps to <pstatefield> SVCRSMZA.

Operation

The description of MSR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.377 SMSUBL

Signed multiply-subtract long

This instruction multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	0 1	Rm	1	Ra	Rn	Rd
sf	op54				U			o0			

Encoding

```
SMSUBL  <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
SMNEGL	Ra == '11111'

Operation

```
constant bits(32) operand1 = X[n, 32];
constant bits(32) operand2 = X[m, 32];
constant bits(64) operand3 = X[a, 64];

constant integer result = SInt(operand3) - (SInt(operand1) * SInt(operand2));

X[d, 64] = result<63:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.378 SMULH

Signed multiply high

This instruction multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	1 0	Rm	0	(1)(1)(1)(1)(1)	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
SMULH  <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
constant bits(64) operand1 = X[n, 64];
constant bits(64) operand2 = X[m, 64];

constant integer result = SInt(operand1) * SInt(operand2);

X[d, 64] = result<127:64>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.379 SMULL

Signed multiply long

This instruction multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding

```
SMULL  <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMADDL  <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational Information

The description of SMADDL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.380 SSBB

Speculative store bypass barrier

This instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions. For more information and details of the semantics, see Speculative Store Bypass Barrier (SSBB).

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31–29	28–26	25–12	11–8	7	6–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	0 0 0 0	1	0 0	1 1 1 1 1
			CRm		opc	Rt

Encoding

```
SSBB
```

is equivalent to

```
DSB  #0
```

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.381 ST2G

Store Allocation Tags

This instruction stores an Allocation Tag to two Tag Granules of memory. 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 source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	1	0	1	imm9		0	1	Rn		Rt	
							opc					op2				

Encoding

```
ST2G  <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	1	0	1	imm9		1	1	Rn		Rt	
							opc					op2				

Encoding

```
ST2G  <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31			28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	1	0	1	imm9	1	0	Rn	Rt
								opc				op2			

Encoding

```
ST2G  <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = FALSE;
constant boolean postindex = FALSE;
```

Assembler Symbols

<Xt|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;
bits(64) address2;
constant bits(64) data = if t == 31 then SP[64] else X[t, 64];
constant bits(4) tag = AArch64.AllocationTagFromAddress(data);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant boolean stzgm = FALSE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

if !postindex then
    address = AddressAdd(address, offset, accdesc);

address2 = AddressIncrement(address, TAG_GRANULE, accdesc);

AArch64.MemTag[address , accdesc] = tag;
AArch64.MemTag[address2, accdesc] = tag;

if writeback then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if postindex then
    address = AddressAdd(address, offset, accdesc);

if n == 31 then
    SP[64] = address;
else
    X[n, 64] = address;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.382 ST64B

Single-copy atomic 64-byte store without status result

This instruction stores eight 64-bit doublewords from consecutive registers to a memory location. The store starts at register `Xt`, with the data being formed as `Data[511:0] = X(t+7)::X(t+6)::X(t+5)::X(t+4)::X(t+3)::X(t+2)::X(t+1)::Xt`. The data is stored atomically and is required to be 64-byte aligned.

It is IMPLEMENTATION DEFINED which memory locations support this instruction. A memory location that supports `ST64B` also supports `LD64B`.

For more information, including about the memory types accessible and how the accesses are performed, see Single-copy atomic 64-byte load/store.

Integer

(FEAT_LS64)

Encoding

```
ST64B  <Xt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64) then EndOfDecode(Decode_UNDEF);
if Rt<4:3> == '11' || Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant boolean withstatus = FALSE;
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean tagchecked = n != 31;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly ST64B.

Assembler Symbols

<Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;

constant AccessDescriptor accdesc = CreateAccDescLS64(MemOp_STORE, withstatus, tagchecked);

for i = 0 to 7
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i : 64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

MemStore64B(address, data, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.383 ST64BV

Single-copy atomic 64-byte store with status result

This instruction stores eight 64-bit doublewords from consecutive registers to a memory location, and writes the status result of the store to a register. The store starts at register `Xt`, with the data being formed as `Data[511:0] = X(t+7)::X(t+6)::X(t+5)::X(t+4)::X(t+3)::X(t+2)::X(t+1)::Xt`. The data is stored atomically and is required to be 64-byte aligned.

It is IMPLEMENTATION DEFINED which memory locations support this instruction. A memory location that supports `ST64BV` also supports `ST64BV0`.

For more information, including about the memory types accessible and how the accesses are performed, see Single-copy atomic 64-byte load/store.

Integer

(FEAT_LS64_V)

Encoding

```
ST64BV  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64_V) then EndOfDecode(Decode_UNDEF);
if Rt<4:3> == '11' || Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant boolean withstatus = TRUE;
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the “Rs” field.

The value returned is:

`0xFFFFFFFF`**_FFFFFFFF** If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

!= `0xFFFFFFFF`**_FFFFFFFF** If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckST64BVEnabled();

bits(512) data;
bits(64) address;
bits(64) value;

constant AccessDescriptor accdesc = CreateAccDescLS64(MemOp_STORE, withstatus, tagchecked);

for i = 0 to 7
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i : 64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(64) status = MemStore64BWithRet(address, data, accdesc);

if s != 31 then X[s, 64] = status;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.384 ST64BV0

Single-copy atomic 64-byte EL0 store with status result

This instruction stores eight 64-bit doublewords from consecutive registers to a memory location, with the bottom 32 bits taken from ACCDATA_EL1, and writes the status result of the store to a register. The store starts at register `Xt`, with the data being formed as `Data[511:0] = X(t+7)::X(t+6)::X(t+5)::X(t+4)::X(t+3)::X(t+2)::X(t+1)::Xt[63:32]::ACCDATA_EL1()[31:0]`. The data is stored atomically and is required to be 64-byte aligned.

It is IMPLEMENTATION DEFINED which memory locations support this instruction. A memory location that supports `ST64BV0` also supports `ST64BV`.

For more information, including about the memory types accessible and how the accesses are performed, see Single-copy atomic 64-byte load/store.

Integer

(FEAT_LS64_ACCDATA)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 12	11	10	9 … 5	4 … 0
1	1	1	1	1	0	0	0	0	0	1	Rs	1	0 1 0	0	0	Rn	Rt
size					VR		A		R			o3	opc				

Encoding

```
ST64BV0  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64_ACCDATA) then EndOfDecode(Decode_UNDEF);
if Rt<4:3> == '11' || Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant boolean withstatus = TRUE;
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the “Rs” field.

The value returned is:

`0xFFFFFFFF`**_FFFFFFFF** If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

!= `0xFFFFFFFF`**_FFFFFFFF** If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckST64BV0Enabled();

bits(512) data;
bits(64) address;
bits(64) value;

constant AccessDescriptor accdesc = CreateAccDescLS64(MemOp_STORE, withstatus, tagchecked);

constant bits(64) Xt = X[t, 64];
value<31:0> = ACCDATA_EL1<31:0>;
value<63:32> = Xt<63:32>;
if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
data<63:0> = value;

for i = 1 to 7
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i : 64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(64) status = MemStore64BWithRet(address, data, accdesc);

if s != 31 then X[s, 64] = status;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.385 STADD, STADDL

Atomic add on word or doubleword, without return

This instruction 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` does not have release semantics.
- `STADDL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STADD  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADD  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STADDL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STADD  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADD  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STADDL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADDL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Operational Information

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.386 STADDB, STADDLB

Atomic add on byte, without return

This instruction 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` does not have release semantics.
- `STADDLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 0 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STADDB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STADDLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.387 STADDH, STADDLH

Atomic add on halfword, without return

This instruction 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` does not have release semantics.
- `STADDLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 0 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STADDH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STADDLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.388 STCLR, STCLRL

Atomic bit clear on word or doubleword, without return

This instruction 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` does not have release semantics.
- `STCLRL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STCLR  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLR  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STCLRL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STCLR  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLR  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STCLRL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.389 STCLRB, STCLRLB

Atomic bit clear on byte, without return

This instruction 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` does not have release semantics.
- `STCLRLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB. This means:

- The encodings in this description are named to match the encodings of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 0 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STCLRB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STCLRLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.390 STCLRH, STCLRLH

Atomic bit clear on halfword, without return

This instruction 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` does not have release semantics.
- `STCLRLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH. This means:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 0 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STCLRH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STCLRLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.391 STEOR, STEORL

Atomic exclusive-OR on word or doubleword, without return

This instruction 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` does not have release semantics.
- `STEORL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEOR, LDEORA, LDEORAL, LDEORL. This means:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STEOR  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEOR  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STEORL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STEOR  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEOR  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STEORL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEORL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational Information

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.392 STEORB, STEORLB

Atomic exclusive-OR on byte, without return

This instruction 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` does not have release semantics.
- `STEORLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STEORB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STEORLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.393 STEORH, STEORLH

Atomic exclusive-OR on halfword, without return

This instruction 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` does not have release semantics.
- `STEORLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH. This means:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STEORH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STEORLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.394 STG

Store Allocation Tag

This instruction stores an Allocation Tag to memory. 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 source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	0	0	1	imm9		0	1	Rn		Rt	
						opc					op2					

Encoding

```
STG  <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	0	0	1	imm9		1	1	Rn		Rt	
						opc					op2					

Encoding

```
STG  <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31			28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	0	0	1	imm9	1	0	Rn	Rt
								opc				op2			

Encoding

```
STG  <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = FALSE;
constant boolean postindex = FALSE;
```

Assembler Symbols

<Xt|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant boolean stzgm = FALSE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(64) data = if t == 31 then SP[64] else X[t, 64];
constant bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, accdesc] = tag;

if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);

    if n == 31 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SP[64] = address;
    else
        X[n, 64] = address;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.395 STGM

Store Allocation Tag multiple

This instruction writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and the Allocation Tag written to address A is taken from the source register at 4*A<7:4>+3:4*A<7:4>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	1	0	1	0 0 0 0 0 0 0 0 0		0	0	Rn		Rt	
						opc			imm9		op2					

Encoding

```
STGM  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;

constant bits(64) data = X[t, 64];
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
constant integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
constant bits(64) curraddress = address;
constant boolean stzgm = FALSE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

for i = 0 to count-1
    constant bits(4) tag = Elem[data, index, 4];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.MemTag[address, accdesc] = tag;
address = AddressIncrement(address, TAG_GRANULE, accdesc);
index = index + 1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.396 STGP

Store Allocation Tag and pair of registers

This instruction 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.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

Encoding

```
STGP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

Encoding

```
STGP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
0	1	1	0	1	0	0	1	0	0	simm7	Rt2	Rn	Rt
opc					VR				L				

Encoding

```
STGP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
constant boolean writeback = FALSE;
constant boolean postindex = FALSE;
```

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index” and “Pre-index” variants: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the “simm7” field.

For the “Signed offset” variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “simm7” field.

Operation

```
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant boolean stzgm = FALSE;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE, stzgm, ispair, t, t2);

if !postindex then
    address = AddressAdd(address, offset, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsAligned(address, TAG_GRANULE) then
    constant FaultRecord fault = AlignmentFault(accdesc, address);
    AArch64.Abort(fault);

constant bits(128) data = (if BigEndian(accdesc.acctype) then X[t, 64]:X[t2, 64]
                                                         else X[t2, 64]:X[t, 64]);
Mem[address, 16, accdesc] = data;

AArch64.MemTag[address, accdesc] = AArch64.AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);

    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.397 STILP

Store-release ordered pair of registers

This instruction calculates an address from a base register value and an optional offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The instruction also has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release, with the additional requirement that:

- When using the pre-index addressing mode, the Memory effects associated with Xt2/Wt2 are Ordered-before the Memory effects associated with Xt1/Wt1.
- For all other addressing modes, the Memory effects associated with Xt1/Wt1 are Ordered-before the Memory effects associated with Xt2/Wt2.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LRCPC3)

Encoding for the 32-bit pre-index variant

Applies when `(size == 10 && opc2 == 0000)`

```
STILP  <Wt1>, <Wt2>, [<Xn|SP>, #-8]!
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc2 == 0001)`

```
STILP  <Wt1>, <Wt2>, [<Xn|SP>]
```

Encoding for the 64-bit pre-index variant

Applies when `(size == 11 && opc2 == 0000)`

```
STILP  <Xt1>, <Xt2>, [<Xn|SP>, #-16]!
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc2 == 0001)`

```
STILP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC3) then EndOfDecode(Decode_UNDEF);
constant boolean ispair = TRUE;
constant boolean wback = opc2<0> == '0';
```

STILP has the same CONSTRAINED UNPREDICTABLE behavior as STP. For information about this CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP and STILP.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant integer{} scale = 2 + UInt(size<0>);
constant integer{} datasize = 8 << scale;
constant integer offset = if opc2<0> == '0' then -1 * (2 << scale) else 0;
constant boolean acqrel = FALSE;
constant boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;   // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked, ispair, acqrel, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];

bits(2*datasize) full_data;
if BigEndian(accdesc.acctype) then
    full_data = data1:data2;
else
    full_data = data2:data1;
accdesc.highestaddressfirst = offset < 0;
Mem[address, 2*dbytes, accdesc] = full_data;

if wback then
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.398 STLLR

Store LORelease register

This instruction 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 addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLLR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLLR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer elsize = 8 << UInt(size);
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
constant integer dbytes = elsize DIV 8;

constant AccessDescriptor accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

Mem[address, dbytes, accdesc] = X[t, elsize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.399 STLLRB

Store LORelease register byte

This instruction 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 addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

Encoding

```
STLLRB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

Mem[address, 1, accdesc] = X[t, 8];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.400 STLLRH

Store LORelease register halfword

This instruction 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 addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

Encoding

```
STLLRH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

Mem[address, 2, accdesc] = X[t, 16];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.401 STLR

Store-release register

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 2 classes: No offset and Pre-index

No offset

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean wback = FALSE;
constant integer offset = 0;
constant boolean rt_unknown = FALSE;

constant integer elsize = 8 << UInt(size);
constant integer datasize = elsize;
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Pre-index

(FEAT_LRCPC3)

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLR  <Wt>, [<Xn|SP>, #-4]!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLR  <Xt>, [<Xn|SP>, #-8]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC3) then EndOfDecode(Decode_UNDEF);
constant boolean wback = TRUE;

constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer offset = -1 * (1 << UInt(size));
constant boolean acquire = FALSE;
constant boolean tagchecked = TRUE;

boolean rt_unknown = FALSE;

if n == t && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;   // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
bits(64) address;
constant integer dbytes = datasize DIV 8;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
bits(datasize) data;
if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];
Mem[address, dbytes, accdesc] = data;
if wback then
    if n == 31 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.402 STLRB

Store-release register byte

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24 23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1	0	0	0 1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size						L		Rs	o0	Rt2		

Encoding

```
STLRB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

Mem[address, 1, accdesc] = X[t, 8];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.403 STLRH

Store-release register halfword

This instruction 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 addressing modes, see Load/Store addressing modes.

Encoding

```
STLRH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

Mem[address, 2, accdesc] = X[t, 16];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.404 STLTXR

Store-release unprivileged exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Note

For the purposes of the Exclusives monitors, and the forward progress guarantees for Load-Exclusive and Store-Exclusive loops, `STLTXR` is equivalent to `STXR`.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

31	30	29						23	22	21	20			16	15	14					10	9				5	4				0
1	sz	0	0	1	0	0	1	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)		Rn					Rt				
									L						o0					Rt2										

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
STLTXR  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
STLTXR  <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant boolean acqrel = TRUE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case c of
        when Constraint_UNKNOWN     rt_unknown = TRUE;     // store UNKNOWN value
        when Constraint_UNDEF       EndOfDecode(Decode_UNDEF);
        when Constraint_NOP         EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN     rn_unknown = TRUE;     // address is UNKNOWN
        when Constraint_UNDEF       EndOfDecode(Decode_UNDEF);
        when Constraint_NOP         EndOfDecode(Decode_NOP);
```

STLTXR has the same CONSTRAINED UNPREDICTABLE behavior as STLXR. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXR.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.405 STLUR

Store-release register (unscaled)

This instruction 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.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29					24	23	22	21	20				12	11	10	9			5	4			0
1	x	0	1	1	0	0	1	0	0	0	imm9					0	0	Rn				Rt			
size								opc																	

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLUR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLUR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);

constant integer datasize = 8 << scale;
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.406 STLURB

Store-release register byte (unscaled)

This instruction 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.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	0	1	1	0	0	1	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
STLURB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);

constant integer datasize = 8;
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.407 STLURH

Store-release register halfword (unscaled)

This instruction 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.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	0	1	1	0	0	1	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
STLURH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);

constant integer datasize = 16;
constant boolean acquire = FALSE;
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.408 STLXP

Store-release exclusive pair of registers

This instruction 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. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. If a 64-bit pair 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 addressing modes, see Load/Store addressing modes.

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
STLXP  <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
STLXP  <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
constant boolean acqrel = TRUE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXP.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;

constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = FALSE;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, ispair, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    constant bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(accdesc.acctype) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.409 STLXR

Store-release exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLXR  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLXR  <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 8 << UInt(size);
constant boolean acqrel = TRUE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXR.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.410 STLXRB

Store-release exclusive register byte

This instruction stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24 23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1	0	0	0 0	0	0	Rs	1	(1) (1) (1) (1) (1)	Rn	Rt
size						L			o0	Rt2		

Encoding

```
STLXRB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = TRUE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRB.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If `AArch64.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.411 STLXRH

Store-release exclusive register halfword

This instruction stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14–10	9–5	4–0
0	1	0	0	1	0	0	0	0	0	0	Rs	1	(1)(1)(1)(1)(1)	Rn	Rt
size									L			o0	Rt2		

Encoding

```
STLXRH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = TRUE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRH.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.412 STNP

Store pair of registers, with non-temporal hint

This instruction 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 addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STNP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
STNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = TRUE;
constant integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
bits(64) address;
constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
constant bits(2*datasize) data = (if BigEndian(accdesc.acctype)
                                  then X[t,  datasize]:X[t2, datasize]
                                  else X[t2, datasize]:X[t,  datasize]);
Mem[address, 2 * dbytes, accdesc] = data;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.413 STP

Store pair of registers

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
STP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
STP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
STP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index 32-bit” and “Pre-index 32-bit” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Post-index 64-bit” and “Pre-index 64-bit” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Signed offset 32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “Signed offset 64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer{} scale = 2 + UInt(opc<1>);
constant integer{} datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
constant boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;   // Value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Value stored is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];

constant bits(2*datasize) data = (if BigEndian(accdesc.acctype) then data1:data2
                                                                else data2:data1);
Mem[address, 2 * dbytes, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.414 STR (immediate)

Store register (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1 1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size				VR			opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STR  <Wt>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STR  <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1 1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size				VR			opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STR  <Wt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STR  <Xt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
1 x	1 1 1	0	0 1	0 0	imm12	Rn	Rt
size		VR		opc			

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STR  <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STR  <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
constant integer scale = UInt(size);
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<pimm>

For the “32-bit” variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the “imm12” field as <pimm>/4.

For the “64-bit” variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer{} datasize = 8 << scale;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean rt_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

bits(datasize) data;
if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];

Mem[address, datasize DIV 8, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.415 STR (register)

Store register (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 0	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STR  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STR  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);            // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer scale = UInt(size);
constant integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

For the “32-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the “64-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer{} datasize = 8 << scale;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.416 STRB (immediate)

Store register byte (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size					VR			opc							

Encoding

```
STRB  <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size					VR			opc							

Encoding

```
STRB  <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	0	1	1	1	0	0	1	0	0	imm12	Rn	Rt
size					VR			opc				

Encoding

```
STRB  <Wt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STRB (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean rt_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

bits(8) data;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

Mem[address, 1, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.417 STRB (register)

Store register byte (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

Encoding for the Extended register variant

Applies when `(option != 011)`

```
STRB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the Shifted register variant

Applies when `(option == 011)`

```
STRB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);            // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
110	SXTW
111	SXTX

<amount>

Is the index shift amount, it must be `#0`, encoded in “S” as 0 if omitted, or as 1 if present.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, 1, accdesc] = X[t, 8];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.418 STRH (immediate)

Store register halfword (immediate)

This instruction 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 addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1	1	0	0	0	0 0	0	imm9	0 1	Rn	Rt
size			VR			opc					

Encoding

```
STRH  <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1	1	0	0	0	0 0	0	imm9	1 1	Rn	Rt
size			VR			opc					

Encoding

```
STRH  <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28	27	26	25	24	23 22	21 … 10	9 … 5	4 … 0
0 1	1 1	1	0	0	1	0 0	imm12	Rn	Rt
size			VR			opc			

Encoding

```
STRH  <Wt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STRH (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the “imm12” field as <pimm>/2.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = wback || n != 31;

Constraint c;
boolean rt_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

bits(16) data;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

Mem[address, 2, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.419 STRH (register)

Store register halfword (register)

This instruction 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 addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31 30	29 28	27	26	25	24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1	1	0	0	0	0 0	1	Rm	option	S	1 0	Rn	Rt
size			VR			opc							

Encoding

```
STRH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);            // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = TRUE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, 2, accdesc] = X[t, 16];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.420 STSET, STSETL

Atomic bit set on word or doubleword, without return

This instruction 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` does not have release semantics.
- `STSETL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSET, LDSETA, LDSETAL, LDSETL. This means:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	R	1	Rs		0	0 1	1	0	0	Rn		1 1 1 1 1	
size				VR			A					o3	opc						Rt	

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STSET  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSET  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STSETL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STSET  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSET  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STSETL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSETL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.421 STSETB, STSETLB

Atomic bit set on byte, without return

This instruction 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` does not have release semantics.
- `STSETLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB. This means:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSETB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSETLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.422 STSETH, STSETLH

Atomic bit set on halfword, without return

This instruction 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` does not have release semantics.
- `STSETLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSETH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSETLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.423 STSHH

Store shared hint

This instruction signals to the memory system that if the next instruction in program order generates an explicit write effect, then it is to a location that one or more other threads of execution will observe, and there is a performance benefit to ensuring that the updated value from the write to that location propagates to those other observers with minimal latency.

The thread of execution on the other observers might be polling the location using load or load-exclusive instructions, or may have executed a `PRFM IR` instruction targeting the location.

System

(FEAT_PCDPHINT)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 1 1 0 (CRm)	0 0 x (op2)	1 1 1 1 1

Encoding

```
STSHH  <policy>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PCDPHINT) then EndOfDecode(Decode_NOP);
constant boolean stream = op2<0> == '1';
```

Assembler Symbols

<policy>

`<policy>` is one of:

KEEP Signals to the memory system that there may be a performance benefit to retaining the updated location in the local cache of the updating PE.

STRM Signals to the memory system that there may be a performance benefit to ensuring the updated location is not retained in the local cache of the updating PE.

op2<0>	<policy>
0	KEEP
1	STRM

Operation

```
Hint_StoreShared(stream);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.424 STSMAX, STSMAXL

Atomic signed maximum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAX` does not have release semantics.
- `STSMAXL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL. This means:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	R	1	Rs		0	1	0 0	0	0	Rn		1 1 1 1 1	
size				VR			A					o3	opc						Rt	

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STSMAX  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STSMAXL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STSMAX  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STSMAXL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.425 STSMAXB, STSMAXLB

Atomic signed maximum on byte, without return

This instruction 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` does not have release semantics.
- `STSMAXLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB. This means:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSMAXB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSMAXLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.426 STSMAXH, STSMAXLH

Atomic signed maximum on halfword, without return

This instruction 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` does not have release semantics.
- `STSMAXLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSMAXH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSMAXLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.427 STSMIN, STSMINL

Atomic signed minimum on word or doubleword, without return

This instruction 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` does not have release semantics.
- `STSMINL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	R	1	Rs		0	1	0 1	0	0	Rn		1 1 1 1 1	
size				VR			A					o3	opc						Rt	

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STSMIN  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STSMINL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STSMIN  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STSMINL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.428 STSMINB, STSMINLB

Atomic signed minimum on byte, without return

This instruction 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` does not have release semantics.
- `STSMINLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB. This means:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSMINB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSMINLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.429 STSMINH, STSMINLH

Atomic signed minimum on halfword, without return

This instruction 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` does not have release semantics.
- `STSMINLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSMINH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSMINLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.430 STTADD, STTADDL

Atomic add unprivileged, without return

This instruction 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.

- `STTADD` does not have release semantics.
- `STTADDL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDTADD, LDTADDA, LDTADDAL, LDTADDL. This means:

- The encodings in this description are named to match the encodings of LDTADD, LDTADDA, LDTADDAL, LDTADDL.
- The description of LDTADD, LDTADDA, LDTADDAL, LDTADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSUI)

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && R == 0)`

```
STTADD  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTADD  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(sz == 0 && R == 1)`

```
STTADDL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTADDL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && R == 0)`

```
STTADD  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTADD  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(sz == 1 && R == 1)`

```
STTADDL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTADDL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDTADD, LDTADDA, LDTADDAL, LDTADDL gives the operational pseudocode for this instruction.

Operational Information

The description of LDTADD, LDTADDA, LDTADDAL, LDTADDL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.431 STTCLR, STTCLRL

Atomic bit clear unprivileged, without return

This instruction 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.

- `STTCLR` does not have release semantics.
- `STTCLRL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL. This means:

- The encodings in this description are named to match the encodings of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL.
- The description of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSUI)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9				5	4				0
0	sz	0	1	1	0	0	1	0	R	1	Rs				0	0	0	1		0	1	Rn					1	1	1	1	1
								A							o3		opc														Rt

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && R == 0)`

```
STTCLR  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTCLR  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(sz == 0 && R == 1)`

```
STTCLRL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTCLRL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && R == 0)`

```
STTCLR  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTCLR  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(sz == 1 && R == 1)`

```
STTCLRL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL gives the operational pseudocode for this instruction.

Operational Information

The description of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.432 STTNP

Store unprivileged pair of registers, with non-temporal hint

This instruction calculates an address from a base register value and an immediate offset, and stores two 64-bit doublewords to the calculated address, from two registers. For information about addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Signed offset

(FEAT_LSUI)

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
1	1	1	0	1	0	0	0	0	0	imm7	Rt2	Rn	Rt
opc					VR				L				

Encoding

```
STTNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);

constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = TRUE;
constant integer datasize = 64;
constant bits(64) offset = LSL(SignExtend(imm7, 64), 3);
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
constant integer dbytes = datasize DIV 8;
constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
constant bits(2*datasize) data = (if BigEndian(accdesc.acctype)
                                  then X[t,  datasize]:X[t2, datasize]
                                  else X[t2, datasize]:X[t,  datasize]);
Mem[address, 2 * dbytes, accdesc] = data;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.433 STTP

Store unprivileged pair of registers

This instruction calculates an address from a base register value and an immediate offset, and stores two 64-bit doublewords to the calculated address, from two registers.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_LSUI)

Encoding

```
STTP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

(FEAT_LSUI)

Encoding

```
STTP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_LSUI)

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
1	1	1	0	1	0	0	1	0	0	imm7	Rt2	Rn	Rt
opc					VR				L				

Encoding

```
STTP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
```

STTP has the same CONSTRAINED UNPREDICTABLE behavior as STP. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Load/store register pair (post-indexed)” and “Load/store register pair (pre-indexed)” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Load/store register pair (offset)” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer{} scale = 2 + UInt(opc<1>);
constant integer{} datasize = 64;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
constant boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;   // Value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Value stored is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Constraint_NOP        EndOfDecode(Decode_NOP);
    end
end
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];

constant bits(2*datasize) data = (if BigEndian(accdesc.acctype) then data1:data2
                                                                else data2:data1);
Mem[address, 2 * dbytes, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.434 STTR

Store register (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	0	0	imm9		1	0	Rn		Rt	
size				VR			opc										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STTR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STTR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 8 << scale;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.435 STTRB

Store register byte (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size					VR			opc							

Encoding

```
STTRB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 8;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.436 STTRH

Store register halfword (unprivileged)

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size					VR			opc							

Encoding

```
STTRH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer{} datasize = 16;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.437 STTSET, STTSETL

Atomic bit set unprivileged, without return

This instruction 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.

- `STTSET` does not have release semantics.
- `STTSETL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDTSET, LDTSETA, LDTSETAL, LDTSETL. This means:

- The encodings in this description are named to match the encodings of LDTSET, LDTSETA, LDTSETAL, LDTSETL.
- The description of LDTSET, LDTSETA, LDTSETAL, LDTSETL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSUI)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9				5	4				0
0	sz	0	1	1	0	0	1	0	R	1	Rs				0	0	1	1		0	1	Rn					1	1	1	1	1
								A							o3	opc															Rt

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && R == 0)`

```
STTSET  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTSET  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(sz == 0 && R == 1)`

```
STTSETL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTSETL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && R == 0)`

```
STTSET  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTSET  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(sz == 1 && R == 1)`

```
STTSETL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTSETL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDTSET, LDTSETA, LDTSETAL, LDTSETL gives the operational pseudocode for this instruction.

Operational Information

The description of LDTSET, LDTSETA, LDTSETAL, LDTSETL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.438 STTXR

Store unprivileged exclusive register

This instruction 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.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Note

For the purposes of the Exclusives monitors, and the forward progress guarantees for Load-Exclusive and Store-Exclusive loops, `STTXR` is equivalent to `STXR`.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
STTXR  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
STTXR  <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

STTXR has the same CONSTRAINED UNPREDICTABLE behavior as STXR. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXR.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.439 STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword, without return

This instruction 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` does not have release semantics.
- `STUMAXL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL. This means:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	R	1	Rs		0	1	1 0	0	0	Rn		1 1 1 1 1	
size				VR			A					o3	opc						Rt	

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STUMAX  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STUMAXL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STUMAX  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STUMAXL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.440 STUMAXB, STUMAXLB

Atomic unsigned maximum on byte, without return

This instruction 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` does not have release semantics.
- `STUMAXLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB. This means:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STUMAXB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STUMAXLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.441 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword, without return

This instruction 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` does not have release semantics.
- `STUMAXLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STUMAXH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STUMAXLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.442 STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword, without return

This instruction 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` does not have release semantics.
- `STUMINL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL. This means:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	R	1	Rs		0	1	1 1	0	0	Rn		1 1 1 1 1	
size				VR			A					o3	opc						Rt	

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STUMIN  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STUMINL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STUMIN  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STUMINL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.443 STUMINB, STUMINLB

Atomic unsigned minimum on byte, without return

This instruction 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` does not have release semantics.
- `STUMINLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB. This means:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 1 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STUMINB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STUMINLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.444 STUMINH, STUMINLH

Atomic unsigned minimum on halfword, without return

This instruction 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` does not have release semantics.
- `STUMINLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STUMINH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STUMINLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.445 STUR

Store register (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size					VR			opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STUR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STUR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 8 << scale;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.446 STURB

Store register byte (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size					VR			opc							

Encoding

```
STURB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 8;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.447 STURH

Store register halfword (unscaled)

This instruction 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 addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1	1	0	0	0	0 0	0	imm9	0 0	Rn	Rt
size			VR			opc					

Encoding

```
STURH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer{} datasize = 16;
constant boolean nontemporal = FALSE;
constant boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                     tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.448 STXP

Store exclusive pair of registers

This instruction 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. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about addressing modes, see Load/Store addressing modes.

31	30	29						23	22	21	20			16	15	14			10	9			5	4			0
1	sz	0	0	1	0	0	0	0	0	1	Rs				0	Rt2				Rn				Rt			
									L						o0												

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
STXP  <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
STXP  <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXP.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;

constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = FALSE;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, ispair, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    constant bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(accdesc.acctype) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.449 STXR

Store exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores.

For information about addressing modes, see Load/Store addressing modes.

31	30	29						23	22	21	20			16	15	14					10	9			5	4			0
1	x	0	0	1	0	0	0	0	0	0	Rs				0	(1)	(1)	(1)	(1)	(1)		Rn				Rt			
	size								L						o0				Rt2										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STXR  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STXR  <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 8 << UInt(size);
constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXR.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.450 STXRB

Store exclusive register byte

This instruction stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24 23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1	0	0	0 0	0	0	Rs	0	(1) (1) (1) (1) (1)	Rn	Rt
size						L			o0	Rt2		

Encoding

```
STXRB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXRB.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If `AArch64.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.451 STXRH

Store exclusive register halfword

This instruction stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about addressing modes, see Load/Store addressing modes.

Encoding

```
STXRH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = FALSE;
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                        tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.452 STZ2G

Store Allocation Tags, zeroing

This instruction 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 base 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.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	1	1	1	imm9		0	1	Rn		Rt	
						opc					op2					

Encoding

```
STZ2G  <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	1	1	1	imm9		1	1	Rn		Rt	
						opc					op2					

Encoding

```
STZ2G  <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31			28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	1	1	1	imm9	1	0	Rn	Rt
								opc				op2			

Encoding

```
STZ2G  <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = FALSE;
constant boolean postindex = FALSE;
```

Assembler Symbols

<Xt|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;
bits(64) address2;
constant bits(64) data = if t == 31 then SP[64] else X[t, 64];
constant bits(4) tag = AArch64.AllocationTagFromAddress(data);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant boolean stzgm = FALSE;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE, stzgm, ispair, t, t);

if !postindex then
    address = AddressAdd(address, offset, accdesc);

address2 = AddressIncrement(address, TAG_GRANULE, accdesc);

if !IsAligned(address, TAG_GRANULE) then
    constant FaultRecord fault = AlignmentFault(accdesc, address);
    AArch64.Abort(fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Mem[address, 2 * TAG_GRANULE, accdesc] = Zeros(TAG_GRANULE * 16);

AArch64.MemTag[address , accdesc] = tag;
AArch64.MemTag[address2, accdesc] = tag;

if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);

    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.453 STZG

Store Allocation Tag, zeroing

This instruction stores an Allocation Tag to memory, zeroing the associated data location. 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 source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	0	1	1	imm9		0	1	Rn		Rt	
						opc					op2					

Encoding

```
STZG  <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	0	1	1	imm9		1	1	Rn		Rt	
						opc					op2					

Encoding

```
STZG  <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = TRUE;
constant boolean postindex = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31			28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	0	1	1	imm9	1	0	Rn	Rt
								opc				op2			

Encoding

```
STZG  <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
constant boolean writeback = FALSE;
constant boolean postindex = FALSE;
```

Assembler Symbols

<Xt|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant boolean stzgm = FALSE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

if !postindex then
    address = AddressAdd(address, offset, accdesc);

if !IsAligned(address, TAG_GRANULE) then
    constant FaultRecord fault = AlignmentFault(accdesc, address);
    AArch64.Abort(fault);

Mem[address, TAG_GRANULE, accdesc] = Zeros(TAG_GRANULE * 8);

constant bits(64) data = if t == 31 then SP[64] else X[t, 64];
constant bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, accdesc] = tag;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);

    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.454 STZGM

Store Allocation Tag and zero multiple

This instruction writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID_EL0.BS, and the Allocation Tag is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)

31			28	27	26	25	24	23	22	21	20									12	11	10	9			5	4			0
1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn				Rt			
								opc			imm9										op2								

Encoding

```
STZGM  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;

constant bits(64) data = X[t, 64];
constant bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant integer size = 4 * (2 ^ (UInt(DCZID_EL0.BS)));
address = Align(address, size);
constant integer count = size >> LOG2_TAG_GRANULE;
constant boolean stzgm = TRUE;
constant AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

for i = 0 to count-1
    AArch64.MemTag[address, accdesc] = tag;
    Mem[address, TAG_GRANULE, accdesc] = Zeros(8*TAG_GRANULE);
    address = AddressIncrement(address, TAG_GRANULE, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.455 SUB (extended register)

Subtract extended and scaled register

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

31	30	29	28				24	23	22	21	20		16	15		13	12		10	9		5	4		0
sf	1	0	0	1	0	1	1	0	0	1	Rm			option			imm3			Rn			Rd		
op		S						opt																	

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
constant integer datasize = 32 << UInt(sf);
constant ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rd” or “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

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

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

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

<amount>

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

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

option	<R>
00x	W

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<m>

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

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = NOT(ExtendReg(m, extend_type, shift, datasize));
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.456 SUB (immediate)

Subtract immediate value

This instruction subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

constant bits(24) imm = if sh == '0' then Zeros(12):imm12 else imm12:Zeros(12);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ZeroExtend(imm, datasize);
bits(datasize) result;
(result, -) = AddWithCarry(operand1, NOT(operand2), '1');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.457 SUB (shifted register)

Subtract optionally-shifted register

This instruction subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register).

31	30	29	28				24	23 22	21	20	16	15	10	9	5	4	0
sf	1	0	0	1	0	1	1	shift	0	Rm		imm6		Rn		Rd	

op S

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

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

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
NEG (shifted register)	Rn == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = NOT(ShiftReg(m, shift_type, shift_amount, datasize));
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, '1');

X[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.458 SUBG

Subtract with tag

This instruction 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.

Integer

(FEAT_MTE)

Encoding

```
SUBG  <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant bits(4) tag_offset = imm4;
constant bits(64) offset = LSL(ZeroExtend(imm6, 64), LOG2_TAG_GRANULE);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<uimm6>

Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the “imm6” field.

<uimm4>

Is an unsigned immediate, in the range 0 to 15, encoded in the “imm4” field.

Operation

```
constant bits(64) operand1 = if n == 31 then SP[64] else X[n, 64];
constant bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
constant bits(16) exclude = GCR_EL1.Exclude;
constant bits(4) rtag = AArch64.ChooseNonExcludedTagOrZero(start_tag, tag_offset, exclude);
bits(64) result;

(result, -) = AddWithCarry(operand1, NOT(offset), '1');

result = AArch64.AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[64] = result;
else
    X[d, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.459 SUBP

Subtract pointer

This instruction 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.

Integer

(FEAT_MTE)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	0	0	1	1 0 1	0 1 1 0	Rm	0 0 0 0 0 0	Rn	Rd
sf		S					opcode		

Encoding

```
SUBP  <Xd>, <Xn|SP>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Rm” field.

Operation

```
bits(64) operand1 = if n == 31 then SP[64] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[64] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = NOT(SignExtend(operand2<55:0>, 64));
bits(64) result;

(result, -) = AddWithCarry(operand1, operand2, '1');

X[d, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.460 SUBPS

Subtract pointer, setting flags

This instruction 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.

This instruction is used by the alias CMPP.

Integer

(FEAT_MTE)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	0	1	1	1 0 1	0 1 1 0	Rm	0 0 0 0 0 0	Rn	Rd
sf		S					opcode		

Encoding

```
SUBPS  <Xd>, <Xn|SP>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMPP	S == '1' && Rd == '11111'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) operand1 = if n == 31 then SP[64] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[64] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = NOT(SignExtend(operand2<55:0>, 64));
bits(64) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '1');

X[d, 64] = result;
PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.461 SUBPT

Subtract checked pointer

This instruction subtracts an optionally-shifted register value from a base address register value, and writes the result to the destination register. The optionally-shifted register value is treated as the offset.

If the operation would have generated a result where the most significant 8 bits of the result register differ from the most significant 8 bits of the base register, then the result is modified such that it is likely to be non-canonical when used as an address.

Integer

(FEAT_CPA)

Encoding

```
SUBPT  <Xd|SP>, <Xn|SP>, <Xm>{, LSL #<amount>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CPA) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the general-purpose destination register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first general-purpose source register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the left shift amount, in the range 0 to 7, defaulting to 0, encoded in the “imm3” field.

Operation

```
bits(64) result;
constant bits(64) base = if n == 31 then SP[64] else X[n, 64];
constant bits(64) offset = LSL(X[m, 64], shift);

result = base - offset;
result = PointerAddCheck(result, base);

if d == 31 then
    SP[64] = result;
else
    X[d, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.462 SUBS (extended register)

Subtract extended and scaled register, setting flags

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

This instruction is used by the alias CMP (extended register).

31	30	29	28				24	23	22	21	20		16	15		13	12		10	9		5	4		0
sf	1	1	0	1	0	1	1	0	0	1	Rm			option			imm3			Rn			Rd		
op		S						opt																	

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
constant integer datasize = 32 << UInt(sf);
constant ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

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

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

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

<amount>

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

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<m>

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

Alias Conditions

Alias	Is preferred when
CMP (extended register)	Rd == '11111'

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = NOT(ExtendReg(m, extend_type, shift, datasize));
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '1');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.463 SUBS (immediate)

Subtract immediate value, setting flags

This instruction subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate).

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

constant bits(24) imm = if sh == '0' then Zeros(12):imm12 else imm12:Zeros(12);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
CMP (immediate)	Rd == '11111'

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ZeroExtend(imm, datasize);
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, NOT(operand2), '1');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.464 SUBS (shifted register)

Subtract optionally-shifted register, setting flags

This instruction subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register) and NEGS.

31	30	29	28 … 24	23 22	21	20 … 16	15 … 10	9 … 5	4 … 0
sf	1	1	0 1 0 1 1	shift	0	Rm	imm6	Rn	Rd
	op	S							

Encoding for the 32-bit variant

Applies when `(sf == 0)`

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

Encoding for the 64-bit variant

Applies when `(sf == 1)`

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

Decode for all variants of this encoding

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

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMP (shifted register)	Rd == '11111'
NEGS	Rn == '11111' && Rd != '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = NOT(ShiftReg(m, shift_type, shift_amount, datasize));
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '1');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.465 SVC

Supervisor call

This instruction 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 syndrome value `0x15`, and the value of the immediate argument.

Encoding

```
SVC  #<imm>
```

Decode for this encoding

```
constant bits(16) imm = imm16;
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
AArch64.CheckForSVCTrap(imm);
AArch64.CallSupervisor(imm);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.466 SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LSE)

Encoding for the 32-bit SWP variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
SWP  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPA variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
SWPA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPAL variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
SWPAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPL variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
SWPL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit SWP variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
SWP  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPA variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
SWPA  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SWPAL variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
SWPAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPL variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
SWPL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);

constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = X[s, datasize];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, regsize] = ZeroExtend(data, regsize);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.467 SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LSE)

Encoding for the SWPB variant

Applies when `(A == 0 && R == 0)`

```
SWPB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPAB variant

Applies when `(A == 1 && R == 0)`

```
SWPAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPALB variant

Applies when `(A == 1 && R == 1)`

```
SWPALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPLB variant

Applies when `(A == 0 && R == 1)`

```
SWPLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);

constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(8) data;
bits(8) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = X[s, 8];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.468 SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory

This instruction 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 neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	1	0	0	0	0	0	Rn	Rt	
size				VR								o3	opc							

Encoding for the SWPH variant

Applies when `(A == 0 && R == 0)`

```
SWPH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPAH variant

Applies when `(A == 1 && R == 0)`

```
SWPAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPALH variant

Applies when `(A == 1 && R == 1)`

```
SWPALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPLH variant

Applies when `(A == 0 && R == 1)`

```
SWPLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);

constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
bits(16) data;
bits(16) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = X[s, 16];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.469 SWPP, SWPPA, SWPPAL, SWPPL

Swap quadword in memory

This instruction atomically loads a 128-bit quadword from a memory location, and stores the value held in a pair of registers back to the same memory location. The value initially loaded from memory is returned in the same pair of registers.

- `SWPPA` and `SWPPAL` load from memory with acquire semantics.
  - `SWPPL` and `SWPPAL` store to memory with release semantics.
  - `SWPP` has neither acquire nor release semantics.

Integer

(FEAT_LSE128)

Encoding for the SWPP variant

Applies when `(A == 0 && R == 0)`

```
SWPP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the SWPPA variant

Applies when `(A == 1 && R == 0)`

```
SWPPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the SWPPAL variant

Applies when `(A == 1 && R == 1)`

```
SWPPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the SWPPL variant

Applies when `(A == 0 && R == 1)`

```
SWPPL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then EndOfDecode(Decode_UNDEF);
if Rt  == '11111' then EndOfDecode(Decode_UNDEF);
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF);

constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly CONSTRAINED UNPREDICTABLE behavior for A64 instructions.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
bits(64) address;
constant bits(64) value1 = X[t, 64];
constant bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                          tagchecked, privileged, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

constant bits(128) comparevalue = bits(128) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64]  = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64]  = data<63:0>;
    X[t2, 64] = data<127:64>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.470 SWPT, SWPTA, SWPTAL, SWPTL

Swap unprivileged

This instruction 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`, `SWPTA` and `SWPTAL` load from memory with acquire semantics.
  - `SWPTL` and `SWPTAL` store to memory with release semantics.
  - `SWPT` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_LSUI)

Encoding for the 32-bit SWPT variant

Applies when `(sz == 0 && A == 0 && R == 0)`

```
SWPT  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPTA variant

Applies when `(sz == 0 && A == 1 && R == 0)`

```
SWPTA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPTAL variant

Applies when `(sz == 0 && A == 1 && R == 1)`

```
SWPTAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPTL variant

Applies when `(sz == 0 && A == 0 && R == 1)`

```
SWPTL  <Ws>, <Wt>, [<Xn|SP>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SWPT variant

Applies when `(sz == 1 && A == 0 && R == 0)`

```
SWPT  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPTA variant

Applies when `(sz == 1 && A == 1 && R == 0)`

```
SWPTA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPTAL variant

Applies when `(sz == 1 && A == 1 && R == 1)`

```
SWPTAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPTL variant

Applies when `(sz == 1 && A == 0 && R == 1)`

```
SWPTL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF);

constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 32 << UInt(sz);
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
constant boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                          tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = X[s, datasize];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, regsize] = ZeroExtend(data, regsize);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.471 SXTB

Signed extend byte

This instruction 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.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0	1 0 0 1 1 0	N	0 0 0 0 0 0	0 0 0 1 1 1	Rn	Rd
	opc			immr	imms		

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SXTB  <Wd>, <Wn>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SXTB  <Xd>, <Wn>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #0, #7
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.472 SXTH

Sign extend halfword

This instruction extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 1 0	N	0 0 0 0 0 0 (immr)	0 0 1 1 1 1 (imms)	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SXTH  <Wd>, <Wn>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SXTH  <Xd>, <Wn>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #0, #15
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.473 SXTW

Sign extend word

This instruction sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding

```
SXTW  <Xd>, <Wn>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #0, #31
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.474 SYS

System instruction

For more information, see Op0 equals `0b01`, cache maintenance, TLB maintenance, and address translation instructions for the encodings of System instructions.

This instruction is used by the aliases APAS, AT, BRB, CFP, COSP, CPP, DC, DVP, GCSPOPCX, GCSPOPX, GCSPUSHM, GCSPUSHX, GCSSS1, IC, TLBI, and TRCIT.

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	op1	CRn	CRm	op2	Rt
			L						

Encoding

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

Decode for this encoding

```
constant integer t       = UInt(Rt);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '01';
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

<Xt>

Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
APAS	`op1 == '110' && CRn == '0111' && CRm == '0000' && op2 == '000'`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
AT	CRn == '0111' && CRm IN {'100x'} && SysOp(op1, '0111', CRm, op2) == Sys_AT
BRB	op1 == '001' && CRn == '0111' && CRm == '0010' && SysOp('001', '0111', '0010', op2) == Sys_BRB
CFP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100'
COSP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '110'
CPP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111'
DC	CRn == '0111' && SysOp(op1, '0111', CRm, op2) == Sys_DC
DVP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'
GCSPOPCX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '101'
GCSPOPX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '110'
GCSPUSHM	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '000'
GCSPUSHX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '100'
GCSSS1	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '010'
IC	CRn == '0111' && SysOp(op1, '0111', CRm, op2) == Sys_IC
TLBI	CRn IN {'100x'} && SysOp(op1, CRn, CRm, op2) == Sys_TLBI
TRCIT	op1 == '011' && CRn == '0111' && CRm == '0010' && op2 == '111'

Operation

```
AArch64.SysInstr(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.475 SYSL

System instruction with result

For more information, see Op0 equals `0b01`, cache maintenance, TLB maintenance, and address translation instructions for the encodings of System instructions.

This instruction is used by the aliases GCSPOPM and GCSSS2.

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	1	0 1	op1	CRn	CRm	op2	Rt
			L						

Encoding

```
SYSL  <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

Decode for this encoding

```
constant integer t       = UInt(Rt);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '01';
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Alias Conditions

Alias	Is preferred when
GCSPOPM	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '001'
GCSSS2	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '011'

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.476 SYSP

128-bit system instruction.

This instruction is used by the alias TLBIP.

System

(FEAT_SYSINSTR128)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 1	0	0 1	op1	CRn	CRm	op2	Rt
			L						

Encoding

```
SYSP  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt1>, <Xt2>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSINSTR128) then EndOfDecode(Decode_UNDEF);
if Rt<0> == '1' && Rt != '11111' then EndOfDecode(Decode_UNDEF);

constant integer t       = UInt(Rt);
constant integer t2      = if t == 31 then 31 else t + 1;
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '01';
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 6, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 8 to 9, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 7, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

<Xt1>

Is the 64-bit name of the first optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second optional general-purpose source register, defaulting to ‘11111’, encoded as “Rt” +1. Defaults to ‘11111’ if “Rt” = ‘11111’.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
TLBIP	CRn IN {'100x'} && SysOp128(op1, CRn, CRm, op2) == Sys_TLBIP

Operation

```
AArch64.SysInstr128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.477 TBNZ

Test bit and branch if nonzero

This instruction 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. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

Encoding

```
TBNZ  <R><t>, #<imm>, <label>
```

Decode for this encoding

```
constant integer t = UInt(Rt);

constant integer datasize = 32 << UInt(b5);
constant integer bit_pos = UInt(b5:b40);
constant bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

<R>

Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an ‘X’ specifier is always permitted, but a ‘W’ specifier is only permitted when the bit number is less than 32.

<t>

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the “Rt” field.

<imm>

Is the bit number to be tested, in the range 0 to 63, encoded in “b5:b40”.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as “imm14” times 4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand = X[t, datasize];
constant boolean branch_conditional = TRUE;
if operand<bit_pos> != '0' then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.478 TBZ

Test bit and branch if zero

This instruction 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. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

Encoding

```
TBZ  <R><t>, #<imm>, <label>
```

Decode for this encoding

```
constant integer t = UInt(Rt);

constant integer datasize = 32 << UInt(b5);
constant integer bit_pos = UInt(b5:b40);
constant bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

<R>

Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an ‘X’ specifier is always permitted, but a ‘W’ specifier is only permitted when the bit number is less than 32.

<t>

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the “Rt” field.

<imm>

Is the bit number to be tested, in the range 0 to 63, encoded in “b5:b40”.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as “imm14” times 4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand = X[t, datasize];
constant boolean branch_conditional = TRUE;
if operand<bit_pos> == '0' then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.479 TLBI

TLB invalidate operation

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0 (L)	0 1	op1	1 0 0 x (CRn)	CRm	op2	Rt

Encoding

```
TLBI  <tlbi_op>{, <Xt>}
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1, CRn, CRm, op2) == Sys_TLBI`.

Assembler Symbols

<tlbi_op>

Is a TLBI operation name, as listed for the TLBI system instruction group, encoded in “op1:CRn:CRm:op2”:

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
000	1000	0001	000	VMALLE1OS	FEAT_TLBIOS
000	1000	0001	001	VAE1OS	FEAT_TLBIOS
000	1000	0001	010	ASIDE1OS	FEAT_TLBIOS
000	1000	0001	011	VAAE1OS	FEAT_TLBIOS
000	1000	0001	101	VALE1OS	FEAT_TLBIOS
000	1000	0001	111	VAALE1OS	FEAT_TLBIOS
000	1000	0010	001	RVAE1IS	FEAT_TLBIRANGE
000	1000	0010	011	RVAAE1IS	FEAT_TLBIRANGE
000	1000	0010	101	RVALE1IS	FEAT_TLBIRANGE
000	1000	0010	111	RVAALE1IS	FEAT_TLBIRANGE
000	1000	0011	000	VMALLE1IS	-
000	1000	0011	001	VAE1IS	-
000	1000	0011	010	ASIDE1IS	-
000	1000	0011	011	VAAE1IS	-
000	1000	0011	101	VALE1IS	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
000	1000	0011	111	VAALE1IS	-
000	1000	0101	001	RVAE1OS	FEAT_TLBIRANGE
000	1000	0101	011	RVAAE1OS	FEAT_TLBIRANGE
000	1000	0101	101	RVALE1OS	FEAT_TLBIRANGE
000	1000	0101	111	RVAALE1OS	FEAT_TLBIRANGE
000	1000	0110	001	RVAE1	FEAT_TLBIRANGE
000	1000	0110	011	RVAAE1	FEAT_TLBIRANGE
000	1000	0110	101	RVALE1	FEAT_TLBIRANGE
000	1000	0110	111	RVAALE1	FEAT_TLBIRANGE
000	1000	0111	000	VMALLE1	-
000	1000	0111	001	VAE1	-
000	1000	0111	010	ASIDE1	-
000	1000	0111	011	VAAE1	-
000	1000	0111	101	VALE1	-
000	1000	0111	111	VAALE1	-
000	1001	0001	000	VMALLE1OSNXS	FEAT_XS
000	1001	0001	001	VAE1OSNXS	FEAT_XS
000	1001	0001	010	ASIDE1OSNXS	FEAT_XS
000	1001	0001	011	VAAE1OSNXS	FEAT_XS
000	1001	0001	101	VALE1OSNXS	FEAT_XS
000	1001	0001	111	VAALE1OSNXS	FEAT_XS
000	1001	0010	001	RVAE1ISNXS	FEAT_XS
000	1001	0010	011	RVAAE1ISNXS	FEAT_XS
000	1001	0010	101	RVALE1ISNXS	FEAT_XS
000	1001	0010	111	RVAALE1ISNXS	FEAT_XS
000	1001	0011	000	VMALLE1ISNXS	FEAT_XS
000	1001	0011	001	VAE1ISNXS	FEAT_XS
000	1001	0011	010	ASIDE1ISNXS	FEAT_XS
000	1001	0011	011	VAAE1ISNXS	FEAT_XS
000	1001	0011	101	VALE1ISNXS	FEAT_XS
000	1001	0011	111	VAALE1ISNXS	FEAT_XS
000	1001	0101	001	RVAE1OSNXS	FEAT_XS
000	1001	0101	011	RVAAE1OSNXS	FEAT_XS
000	1001	0101	101	RVALE1OSNXS	FEAT_XS
000	1001	0101	111	RVAALE1OSNXS	FEAT_XS
000	1001	0110	001	RVAE1NXS	FEAT_XS
000	1001	0110	011	RVAAE1NXS	FEAT_XS
000	1001	0110	101	RVALE1NXS	FEAT_XS
000	1001	0110	111	RVAALE1NXS	FEAT_XS
000	1001	0111	000	VMALLE1NXS	FEAT_XS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
000	1001	0111	001	VAE1NXS	FEAT_XS
000	1001	0111	010	ASIDE1NXS	FEAT_XS
000	1001	0111	011	VAAE1NXS	FEAT_XS
000	1001	0111	101	VALE1NXS	FEAT_XS
000	1001	0111	111	VAALE1NXS	FEAT_XS
100	1000	0000	001	IPAS2E1IS	-
100	1000	0000	010	RIPAS2E1IS	FEAT_TLBIRANGE
100	1000	0000	101	IPAS2LE1IS	-
100	1000	0000	110	RIPAS2LE1IS	FEAT_TLBIRANGE
100	1000	0001	000	ALLE2OS	FEAT_TLBIOS
100	1000	0001	001	VAE2OS	FEAT_TLBIOS
100	1000	0001	100	ALLE1OS	FEAT_TLBIOS
100	1000	0001	101	VALE2OS	FEAT_TLBIOS
100	1000	0001	110	VMALLS12E1OS	FEAT_TLBIOS
100	1000	0010	001	RVAE2IS	FEAT_TLBIRANGE
100	1000	0010	010	VMALLWS2E1IS	FEAT_TLBIW
100	1000	0010	101	RVALE2IS	FEAT_TLBIRANGE
100	1000	0011	000	ALLE2IS	-
100	1000	0011	001	VAE2IS	-
100	1000	0011	100	ALLE1IS	-
100	1000	0011	101	VALE2IS	-
100	1000	0011	110	VMALLS12E1IS	-
100	1000	0100	000	IPAS2E1OS	FEAT_TLBIOS
100	1000	0100	001	IPAS2E1	-
100	1000	0100	010	RIPAS2E1	FEAT_TLBIRANGE
100	1000	0100	011	RIPAS2E1OS	FEAT_TLBIRANGE
100	1000	0100	100	IPAS2LE1OS	FEAT_TLBIOS
100	1000	0100	101	IPAS2LE1	-
100	1000	0100	110	RIPAS2LE1	FEAT_TLBIRANGE
100	1000	0100	111	RIPAS2LE1OS	FEAT_TLBIRANGE
100	1000	0101	001	RVAE2OS	FEAT_TLBIRANGE
100	1000	0101	010	VMALLWS2E1OS	FEAT_TLBIW
100	1000	0101	101	RVALE2OS	FEAT_TLBIRANGE
100	1000	0110	001	RVAE2	FEAT_TLBIRANGE
100	1000	0110	010	VMALLWS2E1	FEAT_TLBIW
100	1000	0110	101	RVALE2	FEAT_TLBIRANGE
100	1000	0111	000	ALLE2	-
100	1000	0111	001	VAE2	-
100	1000	0111	100	ALLE1	-
100	1000	0111	101	VALE2	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
100	1000	0111	110	VMALLS12E1	–
100	1001	0000	001	IPAS2E1ISNXS	FEAT_XS
100	1001	0000	010	RIPAS2E1ISNXS	FEAT_XS
100	1001	0000	101	IPAS2LE1ISNXS	FEAT_XS
100	1001	0000	110	RIPAS2LE1ISNXS	FEAT_XS
100	1001	0001	000	ALLE2OSNXS	FEAT_XS
100	1001	0001	001	VAE2OSNXS	FEAT_XS
100	1001	0001	100	ALLE1OSNXS	FEAT_XS
100	1001	0001	101	VALE2OSNXS	FEAT_XS
100	1001	0001	110	VMALLS12E1OSNXS	FEAT_XS
100	1001	0010	001	RVAE2ISNXS	FEAT_XS
100	1001	0010	010	VMALLWS2E1ISNXS	FEAT_TLBIW
100	1001	0010	101	RVALE2ISNXS	FEAT_XS
100	1001	0011	000	ALLE2ISNXS	FEAT_XS
100	1001	0011	001	VAE2ISNXS	FEAT_XS
100	1001	0011	100	ALLE1ISNXS	FEAT_XS
100	1001	0011	101	VALE2ISNXS	FEAT_XS
100	1001	0011	110	VMALLS12E1ISNXS	FEAT_XS
100	1001	0100	000	IPAS2E1OSNXS	FEAT_XS
100	1001	0100	001	IPAS2E1NXS	FEAT_XS
100	1001	0100	010	RIPAS2E1NXS	FEAT_XS
100	1001	0100	011	RIPAS2E1OSNXS	FEAT_XS
100	1001	0100	100	IPAS2LE1OSNXS	FEAT_XS
100	1001	0100	101	IPAS2LE1NXS	FEAT_XS
100	1001	0100	110	RIPAS2LE1NXS	FEAT_XS
100	1001	0100	111	RIPAS2LE1OSNXS	FEAT_XS
100	1001	0101	001	RVAE2OSNXS	FEAT_XS
100	1001	0101	010	VMALLWS2E1OSNXS	FEAT_TLBIW
100	1001	0101	101	RVALE2OSNXS	FEAT_XS
100	1001	0110	001	RVAE2NXS	FEAT_XS
100	1001	0110	010	VMALLWS2E1NXS	FEAT_TLBIW
100	1001	0110	101	RVALE2NXS	FEAT_XS
100	1001	0111	000	ALLE2NXS	FEAT_XS
100	1001	0111	001	VAE2NXS	FEAT_XS
100	1001	0111	100	ALLE1NXS	FEAT_XS
100	1001	0111	101	VALE2NXS	FEAT_XS
100	1001	0111	110	VMALLS12E1NXS	FEAT_XS
110	1000	0001	000	ALLE3OS	FEAT_TLBIOS
110	1000	0001	001	VAE3OS	FEAT_TLBIOS
110	1000	0001	100	PAALLOS	FEAT_RME

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
110	1000	0001	101	VALE3OS	FEAT_TLBIOS
110	1000	0010	001	RVAE3IS	FEAT_TLBIRANGE
110	1000	0010	101	RVALE3IS	FEAT_TLBIRANGE
110	1000	0011	000	ALLE3IS	-
110	1000	0011	001	VAE3IS	-
110	1000	0011	101	VALE3IS	-
110	1000	0100	011	RPAOS	FEAT_RME
110	1000	0100	111	RPALOS	FEAT_RME
110	1000	0101	001	RVAE3OS	FEAT_TLBIRANGE
110	1000	0101	101	RVALE3OS	FEAT_TLBIRANGE
110	1000	0110	001	RVAE3	FEAT_TLBIRANGE
110	1000	0110	101	RVALE3	FEAT_TLBIRANGE
110	1000	0111	000	ALLE3	-
110	1000	0111	001	VAE3	-
110	1000	0111	100	PAALL	FEAT_RME
110	1000	0111	101	VALE3	-
110	1001	0001	000	ALLE3OSNXS	FEAT_XS
110	1001	0001	001	VAE3OSNXS	FEAT_XS
110	1001	0001	101	VALE3OSNXS	FEAT_XS
110	1001	0010	001	RVAE3ISNXS	FEAT_XS
110	1001	0010	101	RVALE3ISNXS	FEAT_XS
110	1001	0011	000	ALLE3ISNXS	FEAT_XS
110	1001	0011	001	VAE3ISNXS	FEAT_XS
110	1001	0011	101	VALE3ISNXS	FEAT_XS
110	1001	0101	001	RVAE3OSNXS	FEAT_XS
110	1001	0101	101	RVALE3OSNXS	FEAT_XS
110	1001	0110	001	RVAE3NXS	FEAT_XS
110	1001	0110	101	RVALE3NXS	FEAT_XS
110	1001	0111	000	ALLE3NXS	FEAT_XS
110	1001	0111	001	VAE3NXS	FEAT_XS
110	1001	0111	101	VALE3NXS	FEAT_XS

<Xt>

Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.480 TLBIP

TLB invalidate pair operation.

This is an alias of SYSP. This means:

- The encodings in this description are named to match the encodings of SYSP.
- The description of SYSP gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SYSINSTR128)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 1	0	0 1	op1	1 0 0 x	CRm	op2	Rt
			L			CRn			

Encoding

```
TLBIP  <tlbip_op>{, <Xt1>, <Xt2>}
```

is equivalent to

```
SYSP  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt1>, <Xt2>}
```

and is the preferred disassembly when `SysOp128(op1, CRn, CRm, op2) == Sys_TLBIP`.

Assembler Symbols

<tlbip_op>

Is a TLBIP operation name, as listed for the TLBIP system pair instruction group, encoded in "op1:CRn:CRm:op2":

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature
000	1000	0001	001	VAE1OS	FEAT_D128
000	1000	0001	011	VAAE1OS	FEAT_D128
000	1000	0001	101	VALE1OS	FEAT_D128
000	1000	0001	111	VAALE1OS	FEAT_D128
000	1000	0010	001	RVAE1IS	FEAT_D128
000	1000	0010	011	RVAAE1IS	FEAT_D128
000	1000	0010	101	RVALE1IS	FEAT_D128
000	1000	0010	111	RVAALE1IS	FEAT_D128
000	1000	0011	001	VAE1IS	FEAT_D128
000	1000	0011	011	VAAE1IS	FEAT_D128
000	1000	0011	101	VALE1IS	FEAT_D128
000	1000	0011	111	VAALE1IS	FEAT_D128
000	1000	0101	001	RVAE1OS	FEAT_D128

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature
000	1000	0101	011	RVAAE1OS	FEAT_D128
000	1000	0101	101	RVALE1OS	FEAT_D128
000	1000	0101	111	RVAALE1OS	FEAT_D128
000	1000	0110	001	RVAE1	FEAT_D128
000	1000	0110	011	RVAAE1	FEAT_D128
000	1000	0110	101	RVALE1	FEAT_D128
000	1000	0110	111	RVAALE1	FEAT_D128
000	1000	0111	001	VAE1	FEAT_D128
000	1000	0111	011	VAAE1	FEAT_D128
000	1000	0111	101	VALE1	FEAT_D128
000	1000	0111	111	VAALE1	FEAT_D128
000	1001	0001	001	VAE1OSNXS	FEAT_D128
000	1001	0001	011	VAAE1OSNXS	FEAT_D128
000	1001	0001	101	VALE1OSNXS	FEAT_D128
000	1001	0001	111	VAALE1OSNXS	FEAT_D128
000	1001	0010	001	RVAE1ISNXS	FEAT_D128
000	1001	0010	011	RVAAE1ISNXS	FEAT_D128
000	1001	0010	101	RVALE1ISNXS	FEAT_D128
000	1001	0010	111	RVAALE1ISNXS	FEAT_D128
000	1001	0011	001	VAE1ISNXS	FEAT_D128
000	1001	0011	011	VAAE1ISNXS	FEAT_D128
000	1001	0011	101	VALE1ISNXS	FEAT_D128
000	1001	0011	111	VAALE1ISNXS	FEAT_D128
000	1001	0101	001	RVAE1OSNXS	FEAT_D128
000	1001	0101	011	RVAAE1OSNXS	FEAT_D128
000	1001	0101	101	RVALE1OSNXS	FEAT_D128
000	1001	0101	111	RVAALE1OSNXS	FEAT_D128
000	1001	0110	001	RVAE1NXS	FEAT_D128
000	1001	0110	011	RVAAE1NXS	FEAT_D128
000	1001	0110	101	RVALE1NXS	FEAT_D128
000	1001	0110	111	RVAALE1NXS	FEAT_D128
000	1001	0111	001	VAE1NXS	FEAT_D128
000	1001	0111	011	VAAE1NXS	FEAT_D128
000	1001	0111	101	VALE1NXS	FEAT_D128
000	1001	0111	111	VAALE1NXS	FEAT_D128
100	1000	0000	001	IPAS2E1IS	FEAT_D128
100	1000	0000	010	RIPAS2E1IS	FEAT_D128
100	1000	0000	101	IPAS2LE1IS	FEAT_D128
100	1000	0000	110	RIPAS2LE1IS	FEAT_D128
100	1000	0001	001	VAE2OS	FEAT_D128

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature
100	1000	0001	101	VALE2OS	FEAT_D128
100	1000	0010	001	RVAE2IS	FEAT_D128
100	1000	0010	101	RVALE2IS	FEAT_D128
100	1000	0011	001	VAE2IS	FEAT_D128
100	1000	0011	101	VALE2IS	FEAT_D128
100	1000	0100	000	IPAS2E1OS	FEAT_D128
100	1000	0100	001	IPAS2E1	FEAT_D128
100	1000	0100	010	RIPAS2E1	FEAT_D128
100	1000	0100	011	RIPAS2E1OS	FEAT_D128
100	1000	0100	100	IPAS2LE1OS	FEAT_D128
100	1000	0100	101	IPAS2LE1	FEAT_D128
100	1000	0100	110	RIPAS2LE1	FEAT_D128
100	1000	0100	111	RIPAS2LE1OS	FEAT_D128
100	1000	0101	001	RVAE2OS	FEAT_D128
100	1000	0101	101	RVALE2OS	FEAT_D128
100	1000	0110	001	RVAE2	FEAT_D128
100	1000	0110	101	RVALE2	FEAT_D128
100	1000	0111	001	VAE2	FEAT_D128
100	1000	0111	101	VALE2	FEAT_D128
100	1001	0000	001	IPAS2E1ISNXS	FEAT_D128
100	1001	0000	010	RIPAS2E1ISNXS	FEAT_D128
100	1001	0000	101	IPAS2LE1ISNXS	FEAT_D128
100	1001	0000	110	RIPAS2LE1ISNXS	FEAT_D128
100	1001	0001	001	VAE2OSNXS	FEAT_D128
100	1001	0001	101	VALE2OSNXS	FEAT_D128
100	1001	0010	001	RVAE2ISNXS	FEAT_D128
100	1001	0010	101	RVALE2ISNXS	FEAT_D128
100	1001	0011	001	VAE2ISNXS	FEAT_D128
100	1001	0011	101	VALE2ISNXS	FEAT_D128
100	1001	0100	000	IPAS2E1OSNXS	FEAT_D128
100	1001	0100	001	IPAS2E1NXS	FEAT_D128
100	1001	0100	010	RIPAS2E1NXS	FEAT_D128
100	1001	0100	011	RIPAS2E1OSNXS	FEAT_D128
100	1001	0100	100	IPAS2LE1OSNXS	FEAT_D128
100	1001	0100	101	IPAS2LE1NXS	FEAT_D128
100	1001	0100	110	RIPAS2LE1NXS	FEAT_D128
100	1001	0100	111	RIPAS2LE1OSNXS	FEAT_D128
100	1001	0101	001	RVAE2OSNXS	FEAT_D128
100	1001	0101	101	RVALE2OSNXS	FEAT_D128
100	1001	0110	001	RVAE2NXS	FEAT_D128

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature
100	1001	0110	101	RVALE2NXS	FEAT_D128
100	1001	0111	001	VAE2NXS	FEAT_D128
100	1001	0111	101	VALE2NXS	FEAT_D128
110	1000	0001	001	VAE3OS	FEAT_D128
110	1000	0001	101	VALE3OS	FEAT_D128
110	1000	0010	001	RVAE3IS	FEAT_D128
110	1000	0010	101	RVALE3IS	FEAT_D128
110	1000	0011	001	VAE3IS	FEAT_D128
110	1000	0011	101	VALE3IS	FEAT_D128
110	1000	0101	001	RVAE3OS	FEAT_D128
110	1000	0101	101	RVALE3OS	FEAT_D128
110	1000	0110	001	RVAE3	FEAT_D128
110	1000	0110	101	RVALE3	FEAT_D128
110	1000	0111	001	VAE3	FEAT_D128
110	1000	0111	101	VALE3	FEAT_D128
110	1001	0001	001	VAE3OSNXS	FEAT_D128
110	1001	0001	101	VALE3OSNXS	FEAT_D128
110	1001	0010	001	RVAE3ISNXS	FEAT_D128
110	1001	0010	101	RVALE3ISNXS	FEAT_D128
110	1001	0011	001	VAE3ISNXS	FEAT_D128
110	1001	0011	101	VALE3ISNXS	FEAT_D128
110	1001	0101	001	RVAE3OSNXS	FEAT_D128
110	1001	0101	101	RVALE3OSNXS	FEAT_D128
110	1001	0110	001	RVAE3NXS	FEAT_D128
110	1001	0110	101	RVALE3NXS	FEAT_D128
110	1001	0111	001	VAE3NXS	FEAT_D128
110	1001	0111	101	VALE3NXS	FEAT_D128

<Xt1>

Is the 64-bit name of the first optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second optional general-purpose source register, defaulting to ‘11111’, encoded as “Rt” +1. Defaults to ‘11111’ if “Rt” = ‘11111’.

Operation

The description of SYSP gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.481 TRCIT

Trace instrumentation

This instruction generates an instrumentation trace packet that contains the value of the provided register.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_ITE)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 0	1 1 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
TRCIT  <Xt>
```

is equivalent to

```
SYS  #3, C7, C2, #7, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.482 TSB

Trace synchronization barrier

This instruction is a barrier that synchronizes the trace operations of instructions, see Trace Synchronization Barrier (TSB).

If FEAT_TRF is not implemented, this instruction executes as a `NOP`.

System

(FEAT_TRF)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	0 1 0 (op2)	1 1 1 1 1

Encoding

```
TSB  CSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then EndOfDecode(Decode_NOP);
```

Operation

```
if IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_TRBEv1p1) then
    constant boolean trap_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                    !IsInHost() &&
                                    (!HaveEL(EL3) || SCR_EL3.FGTEn2 == '1') &&
                                    HFGITR2_EL2.TSBCSYNC == '1');
    if trap_to_el2 then
        ExceptionRecord except = ExceptionSyndrome(Exception_LDST64BTrap); // to be renamed
        except.syndrome.iss = \texttt{0x4}<24:0>;
        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        constant integer vect_offset = \texttt{0x0};
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);

TraceSynchronizationBarrier();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.483 TST (immediate)

Test bits (immediate)

This instruction performs a bitwise AND of a register value and an immediate value, and discards the results. It updates the condition flags based on the result.

This is an alias of ANDS (immediate). This means:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 1	1 0 0 1 0 0	N	immr	imms	Rn	1 1 1 1 1
	opc						Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
TST  <Wn>, #<imm>
```

is equivalent to

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

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
TST  <Xn>, #<imm>
```

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of ANDS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ANDS (immediate) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.484 TST (shifted register)

Test (shifted register)

This instruction 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.

This is an alias of ANDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 1 (opc)	0 1 0 1 0	shift	0 (N)	Rm	imm6	Rn	1 1 1 1 1 (Rd)

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
TST  <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS  WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
TST  <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS  XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ANDS (shifted register) gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.485 UBFIZ

Unsigned bitfield insert in zeros

This instruction copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	1 0 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
UBFIZ  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #(-<lsb>  MOD  32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
UBFIZ  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM  <Xd>, <Xn>, #(-<lsb>  MOD  64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.486 UBFM

Unsigned bitfield move

This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If `<imms>` is greater than or equal to `<immr>`, this copies a bitfield of (`<imms>`-`<immr>`+1) bits starting from bit position `<immr>` in the source register to the least significant bits of the destination register.

If `<imms>` is less than `<immr>`, this copies a bitfield of (`<imms>`+1) bits from the least significant bits of the source register to bit position (regsize-`<immr>`) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd

opc

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
UBFM  <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
UBFM  <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF);
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer s = UInt(imms);
constant integer r = UInt(immr);

bits(datasize) wmask;
bits(datasize) tmask;
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<immr>

For the “32-bit” variant: is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imms>

For the “32-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Alias	Of variant	Is preferred when
LSL (immediate)	32-bit	imms != '011111' && UInt(imms) + 1 == UInt(immr)
LSL (immediate)	64-bit	imms != '111111' && UInt(imms) + 1 == UInt(immr)
LSR (immediate)	32-bit	imms == '011111'
LSR (immediate)	64-bit	imms == '111111'
UBFIZ		UInt(imms) < UInt(immr)
UBFX		BFXPreferred(sf, opc<1>, imms, immr)
UXTB		immr == '000000' && imms == '000111'
UXTH		immr == '000000' && imms == '001111'

Operation

```
constant bits(datasize) src = X[n, datasize];

// Perform bitfield move on low bits
constant bits(datasize) bot = ROR(src, r) AND wmask;

// Combine extension bits and result bits
X[d, datasize] = bot AND tmask;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.487 UBFX

Unsigned bitfield extract

This instruction copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	1 0 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
UBFX  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc[1], imms, immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
UBFX  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc[1], imms, immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.488 UDF

Permanently undefined

This instruction generates an Undefined Instruction exception (ESR_ELx.EC = `0b000000`). The encodings for `UDF` used in this section are defined as permanently UNDEFINED.

31	30	29	28	25	24	16	15	0
0	0	0	0 0 0 0		0 0 0 0 0 0 0 0 0		imm16	

Encoding

```
UDF  #<imm>
```

Decode for this encoding

```
// The imm16 field is ignored by hardware.
EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<imm>

is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field. The PE ignores the value of this constant.

Operation

```
// No operation.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.489 UDIV

Unsigned divide

This instruction divides the first unsigned source register value by the second unsigned source register value, and writes the result to the destination register. Dividing by zero writes the value zero to the destination register. The condition flags are not affected.

31	30	29	28	21	20	16	15	11	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 0 0 0 1		0	Rn		Rd	
		S							o1				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UDIV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UDIV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant integer dividend = UInt(operand1);
constant integer divisor  = UInt(operand2);
integer result;
if divisor == 0 then
    result = 0;
else
    result = dividend DIV divisor;
X[d, datasize] = result<datasize-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.490 UMADDL

Unsigned multiply-add long

This instruction multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	0 1	Rm	0	Ra	Rn	Rd
sf	op54				U			o0			

Encoding

```
UMADDL  <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
UMULL	Ra == '11111'

Operation

```
constant bits(32) operand1 = X[n, 32];
constant bits(32) operand2 = X[m, 32];
constant bits(64) operand3 = X[a, 64];

constant integer result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d, 64] = result<63:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.491 UMAX (immediate)

Unsigned maximum (immediate)

This instruction determines the unsigned maximum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28	22	21	18	17	10	9	5	4	0
sf	0	0	1 0 0 0 1 1 1		0 0 0 1		imm8		Rn		Rd	
	op	S			opc							

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UMAX  <Wd>, <Wn>, #<uimm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UMAX  <Xd>, <Xn>, #<uimm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer imm = UInt(imm8);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<uimm>

Is an unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = Max(UInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.492 UMAX (register)

Unsigned maximum (register)

This instruction determines the unsigned maximum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28							21	20			16	15						10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm				0	1	1	0	0	1		Rn					Rd				
		S													opcode																

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UMAX  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UMAX  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant integer result = Max(UInt(operand1), UInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.493 UMIN (immediate)

Unsigned minimum (immediate)

This instruction determines the unsigned minimum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28	22	21	18	17	10	9	5	4	0
sf	0	0	1 0 0 0 1 1 1		0 0 1 1		imm8		Rn		Rd	
	op	S				opc						

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UMIN  <Wd>, <Wn>, #<uimm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UMIN  <Xd>, <Xn>, #<uimm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer imm = UInt(imm8);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<uimm>

Is an unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = Min(UInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.494 UMIN (register)

Unsigned minimum (register)

This instruction determines the unsigned minimum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	1	0	1	1		Rn					Rd				
		S																	opcode													

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UMIN  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UMIN  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant integer result = Min(UInt(operand1), UInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.495 UMNEGL

Unsigned multiply-negate long

This instruction multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of UMSUBL. This means:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	0 1	Rm	1	1 1 1 1 1	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
UMNEGL  <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMSUBL  <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational Information

The description of UMSUBL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.496 UMSUBL

Unsigned multiply-subtract long

This instruction multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	0 1	Rm	1	Ra	Rn	Rd
sf	op54				U			o0			

Encoding

```
UMSUBL  <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
UMNEGL	Ra == '11111'

Operation

```
constant bits(32) operand1 = X[n, 32];
constant bits(32) operand2 = X[m, 32];
constant bits(64) operand3 = X[a, 64];

constant integer result = UInt(operand3) - (UInt(operand1) * UInt(operand2));

X[d, 64] = result<63:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.497 UMULH

Unsigned multiply high

This instruction multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	1 0	Rm	0	(1)(1)(1)(1)(1)	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
UMULH  <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
constant bits(64) operand1 = X[n, 64];
constant bits(64) operand2 = X[m, 64];

constant integer result = UInt(operand1) * UInt(operand2);

X[d, 64] = result<127:64>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.498 UMULL

Unsigned multiply long

This instruction multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of UMADDL. This means:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding

```
UMULL  <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMADDL  <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational Information

The description of UMADDL gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.499 UXTB

Unsigned extend byte

This instruction 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.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding

```
UXTB  <Wd>, <Wn>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.500 UXTH

Unsigned extend halfword

This instruction 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.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding

```
UXTH  <Wd>, <Wn>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.501 WFE

Wait for event

This instruction is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see Wait For Event mechanism and Send event.

As described in Wait For Event mechanism and Send event, the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	0 1 0 (op2)	1 1 1 1 1

Encoding

```
WFE
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_WFE();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.502 WFET

Wait for event with timeout

This instruction provides a hint that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see Wait For Event mechanism and Send event.

As described in Wait For Event mechanism and Send event, the execution of a `WFET` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System

(FEAT_WFxT)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 0 1	0 0 0 0 (CRm)	0 0 0 (op2)	Rd

Encoding

```
WFET  <Xt>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_WFxT) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rd” field.

Operation

```
constant integer localtimeout = UInt(X[d, 64]);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    ExecuteAsNOP();

Hint_WFET(localtimeout);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.503 WFI

Wait for interrupt

This instruction is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see Wait For Interrupt.

As described in Wait For Interrupt, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	0 1 1 (op2)	1 1 1 1 1

Encoding

```
WFI
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_WFI();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.504 WFIT

Wait for interrupt with timeout

This instruction provides a hint that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see Wait For Interrupt.

As described in Wait For Interrupt, the execution of a `WFIT` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System

(FEAT_WFxT)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 0 1	0 0 0 0 (CRm)	0 0 1 (op2)	Rd

Encoding

```
WFIT  <Xt>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_WFxT) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rd” field.

Operation

```
constant integer localtimeout = UInt(X[d, 64]);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    ExecuteAsNOP();

Hint_WFIT(localtimeout);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.505 XAFLAG

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.

System

(FEAT_FlagM2)

31 29	28 26	25 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0	0 0 0	0 1 0 0	(0) (0) (0) (0)	0 0 1	1 1 1 1 1
			op1		CRm	op2	Rt

Encoding

```
XAFLAG
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM2) then EndOfDecode(Decode_UNDEF);
```

Operation

```
constant bit n = NOT(PSTATE.C) AND NOT(PSTATE.Z);
constant bit z = PSTATE.Z AND PSTATE.C;
constant bit c = PSTATE.C OR PSTATE.Z;
constant bit v = NOT(PSTATE.C) AND PSTATE.Z;

PSTATE.N = n;
PSTATE.Z = z;
PSTATE.C = c;
PSTATE.V = v;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.506 XPACD, XPACI, XPACLRI

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 `XPACLRI`.

The `XPACD` instruction is used for data addresses, and `XPACI` and `XPACLRI` are used for instruction addresses.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

31	30	29	28							21	20					16	15					11	10	9					5	4				0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1		0	1	0	0	0		D	1	1	1	1	1		Rd				
sf																opcode2												Rn						

Encoding for the XPACD variant

Applies when `(D == 1)`

```
XPACD  <Xd>
```

Encoding for the XPACI variant

Applies when `(D == 0)`

```
XPACI  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant boolean data = (D == '1');
```

System

(FEAT_PAuth)

31		29	28		26	25											12	11				8	7		5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1
																				CRm					op2					

Encoding

```
XPACLRI
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
constant integer d = 30;
constant boolean data = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

```
X[d, 64] = Strip(X[d, 64], data);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.507 YIELD

Yield

This instruction is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see The YIELD instruction.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	0 0 1 (op2)	1 1 1 1 1

Encoding

```
YIELD
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_Yield();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C7
A64 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes the A64 Advanced SIMD and floating-point instructions.

It contains the following sections:

- About the A64 Advanced SIMD and floating-point instructions.
- Alphabetical list of A64 Advanced SIMD and floating-point instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.1 About the A64 Advanced SIMD and floating-point instructions

Alphabetical list of A64 Advanced SIMD and floating-point instructions gives full descriptions of the A64 instructions that are in the following instruction groups:

- Loads and store instructions associated with the SIMD and floating-point registers.
- Data processing instructions with SIMD and floating-point registers.

A64 instruction set encoding in the A64 Instruction Encodings chapter provides an overview of the instruction encodings as part of an instruction class within a functional group.

The rest of this section is a general description of the SIMD and floating-point instructions. It contains the following subsections:

- Register size.
- Output element control.
- Data types.
- Condition flags and related instructions.

C7.1.1 Register size

A64 provides a comprehensive set of packed Single Instruction Multiple Data (SIMD) and scalar operations using data held in the 32 entry 128-bit wide SIMD and floating-point register file.

Each SIMD and floating-point register can be used to hold:

- A single scalar value of the floating-point or integer type.
- A 64-bit wide vector containing one or more elements.
- A 128-bit wide vector containing two or more elements.

Where the entire 128-bit wide register is not fully utilized, the vector or scalar quantity is held in the least significant bits of the register, with the most significant bits being cleared to zero on a write, see Advanced SIMD vector formats.

The following instructions can insert data into individual elements within a SIMD and floating-pointer register without clearing the remaining bits to zero:

- Insert vector element from another vector element or general-purpose register, `INS`.
- Load structure into a single lane, for example `LD3`.
- All second-part narrowing operations, for example `SHRN2`.

C7.1.2 Output element control

When FEAT_AFP is implemented, the FPCR.NEP bit controls how output elements are determined for the scalar Advanced SIMD instructions for elements other than the lowest element of the vector.

If FPCR.NEP == 1, the following instructions determine output elements as follows:

- The 3-input floating-point scalar versions of FMLA (by element) and FMLS (by element) take output elements other than the lowest element from the <Hd>, <Sd>, or <Dd> register.
- The 3-input floating-point FMADD, FMSUB, FNMADD, and FNMSUB instructions take output elements other than the lowest element from the <Ha>, <Sa>, or <Da> register.
- The 2-input floating-point scalar versions of FCMGE (register), FCMGT (register), FCMEQ (register), FACGE, FACGT, take output elements other than the lowest element from the <Hm>, <Sm>, or <Dm> register.
- The 2-input floating-point scalar versions of FMULX, FRECPS, FRSQRTS, FABD, FMUL (by element), FMUL (scalar), FDIV (scalar), FADD (scalar), FSUB (scalar), FMAX (scalar), FMIN (scalar), FMAXNM (scalar), FMINNM (scalar), FNMUL (scalar), take output elements other than the lowest element from the <Hn>, <Sn>, or <Dn> register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For 1-input floating-point scalar versions of the instructions FCVTNS (vector), FCVTMS (vector), FCVTAS (vector), FCVTPS (vector), SCVTF (vector, integer), UCVTF (vector, integer) FCVTZS (vector, integer), FCVTZU (vector, integer), FCVTNU (vector), FCVTMU (vector), FCVTAU (vector), FCVTPU (vector), SCVTF (vector, fixed-point), UCVTF (vector, fixed-point), FCVTZS (vector, fixed-point), FCVTZU (vector, fixed-point), SCVTF (scalar, integer), UCVTF (scalar, integer), SCVTF (scalar, fixed-point), UCVTF (scalar, fixed-point), BFCVT, FCVT, `FCVTXN`, FRECPE, FRECPX, FRSQRTE, FABS (scalar), FNEG (scalar), FSQRT (scalar), FRINTN (scalar), FRINTP (scalar), FRINTM (scalar), FRINTZ (scalar), FRINTA (scalar), FRINTI (scalar), FRINTX (scalar), FRINT32Z (scalar), FRINT32X (scalar), FRINT64Z (scalar), FRINT64X (scalar), take output elements other than the lowest element from the <Hd>, <Sd>, or <Dd> register.

C7.1.3 Data types

The A64 instruction set provides support for arithmetic, conversion, and bitwise operations on:

- Half-precision, single-precision, and double-precision floating-points.
- Signed and unsigned integers.
- Polynomials over {0, 1}.
- When FEAT_FCMA is implemented, complex numbers.

For all AArch64 floating-point operations, including SIMD operations, the rounding mode and exception trap handling are controlled by the FPCR.

Note

- AArch32 Advanced SIMD operations always use Arm standard floating-point arithmetic, regardless of the rounding mode specified by the AArch64 FPCR or the AArch32 FPSCR.
- In AArch64 state, floating-point multiply-add operations are always performed as fused operations, but AArch32 state provides both fused and chained multiply-add instructions.

In addition to operations that consume and produce values of the same width and type, the A64 instruction set supports SIMD and scalar operations that produce a wider or narrower vector result:

- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the A64 instruction set provides a second-part operation, for example `SHRN2`, that can pack the result of a second operation into the upper part of the same destination register.
- Where a SIMD operation widens a 64-bit vector to a 128-bit vector, the A64 instruction set provides a second-part operation, for example `SMLAL2`, that can extract the source from the upper 64 bits of the source registers.

All SIMD operations that could produce side-effects that are not limited to the destination SIMD and floating-point register, for example a potential update of FPSR.Q or FPSR.IDC, have a dedicated scalar variant to support the use of SIMD with loops requiring specialized head or tail handling, or both.

C7.1.4 Condition flags and related instructions

The A64 instruction set provides support for flag setting and conditional operations on the SIMD and floating-point register file:

- Floating-point `FCSEL` and `FCCMP` instructions are equivalent to the integer `CSEL` and `CCMP` instructions.
- Floating-point `FCMP`, `FCMPE`, `FCCMP`, and `FCCMP` instructions set the PSTATE.{N, Z, C, V} flags based on the result of the floating-point comparison.
- Floating-point `FJCVTZS` instruction sets the PSTATE.Z flag if the result of the conversion, when converted back to a double-precision floating-point number, gives precisely the same value as the original. Other PSTATE flags are cleared by this instruction.
- Floating-point and integer instructions provide a means of producing either a scalar or a vector mask based on a comparison in a SIMD and floating-point register, for example `FCMEQ`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

`FCMP` and `FCMPE` differ from the A32/T32 `VCMP` and `VCMPE` instructions, which use the dedicated FPSCR.NZCV field for the result. A64 instructions store the result of an `FCMP` or `FCMPE` operation in the PSTATE.{N, Z, C, V} field.
If FEAT_FlagM2 is implemented, base instructions XAFLAG and AXFLAG convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format. See Table C6-1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions

This section lists every section in the Advanced SIMD and floating-point categories of the A64 instruction set. For details of the format used, see Structure of the A64 assembler language.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.1 ABS

Absolute value (vector)

This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

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

Encoding

```
ABS  D<d>, D<n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

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

Encoding

```
ABS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.2 ADD (vector)

Add (vector)

This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	Rm		1	0	0	0	0	1	Rn		Rd	
		U						size					opcode									

Encoding

```
ADD  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	Rm		1	0	0	0	0	1	Rn		Rd	
		U											opcode									

Encoding

```
ADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 + element2;
V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.3 ADDHN, ADDHN2

Add returning high narrow

This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&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&FP register.

The results are truncated. For rounded results, see RADDHN.

The `ADDHN` instruction writes the vector to the lower half of the destination register and clears the upper half. 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.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1	1	0	size	1	Rm	0 1	0	0	0 0	Rn	Rd
		U									o1				

Encoding

```
ADDHN{2}  <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean round = FALSE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    sum = element1 + element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.4 ADDP (scalar)

Add pair of elements (scalar)

This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

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

Encoding

```
ADDP  D<d>, <Vn>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 64;
constant integer datasize = 128;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
V[d, esize] = IntReduce(ReduceOp_ADD, operand, esize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.5 ADDP (vector)

Add pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&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&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

Encoding

```
ADDP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.6 ADDV

Add across vector

This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 1 0 0 0	1 1 0 1 1	1 0	Rn	Rd
		U						opcode			

Encoding

```
ADDV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '100' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
V[d, esize] = IntReduce(ReduceOp_ADD, operand, esize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.7 AESD

AES single round decryption.

Advanced SIMD

(FEAT_AES)

31–28	27–25	24	23–22	21–17	16–13	12	11–10	9–5	4–0
0 1 0 0	1 1 1	0	0 0 (size)	1 0 1 0 0	0 0 1 0	1 (D)	1 0	Rn	Rd

Encoding

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

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[d, 128];
constant bits(128) operand2 = V[n, 128];
bits(128) result = operand1 EOR operand2;
result = AESInvShiftRows(result);
V[d, 128] = AESInvSubBytes(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.8 AESE

AES single round encryption.

Advanced SIMD

(FEAT_AES)

31	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0 1 0 0		1 1 1		0	0	0	1 0 1 0 0		0 0 1 0		0	1	0	Rn		Rd	
					size						D						

Encoding

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

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[d, 128];
constant bits(128) operand2 = V[n, 128];
bits(128) result = operand1 EOR operand2;
result = AESShiftRows(result);
V[d, 128] = AESSubBytes(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.9 AESIMC

AES inverse mix columns.

Advanced SIMD

(FEAT_AES)

31	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0 1 0 0		1 1 1		0	0	0	1 0 1 0 0		0 0 1 1		1	1	0	Rn		Rd	
					size						D						

Encoding

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

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand = V[n, 128];
V[d, 128] = AESInvMixColumns(operand);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.10 AESMC

AES mix columns.

Advanced SIMD

(FEAT_AES)

31	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0 1 0 0		1 1 1		0	0	0	1 0 1 0 0		0 0 1 1		0	1	0	Rn		Rd	
					size						D						

Encoding

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

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand = V[n, 128];
V[d, 128] = AESMixColumns(operand);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.11 AND (vector)

Bitwise AND (vector)

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

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

Encoding

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

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
V[d, datasize] = operand1 AND operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.12 BCAX

Bit clear and exclusive-OR

This instruction performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive-OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA3)

Encoding

```
BCAX  <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Va>

Is the name of the third SIMD&FP source register, encoded in the “Ra” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[m, 128];
constant bits(128) operand2 = V[n, 128];
constant bits(128) operand3 = V[a, 128];

V[d, 128] = operand2 EOR (operand1 AND NOT(operand3));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.13 BF1CVTL, BF1CVTL2, BF2CVTL, BF2CVTL2

8-bit floating-point convert to BFloat16 (vector)

This instruction converts each 8-bit floating-point element from the lower or upper half of the source vector to BFloat16 while downscaling the value, and places the results in the 16-bit elements of the destination vector. `BF1CVTL` and `BF2CVTL` convert the elements from the lower half of the source vector while scaling the values by $2^{-\text{UInt(FPMR.LSCALE[5:0])}}$ and $2^{-\text{UInt(FPMR.LSCALE2[5:0])}}$, respectively. `BF1CVTL2` and `BF2CVTL2` convert the elements from the upper half of the source vector while scaling the values by $2^{-\text{UInt(FPMR.LSCALE[5:0])}}$ and $2^{-\text{UInt(FPMR.LSCALE2[5:0])}}$, respectively.

The 8-bit floating-point encoding format for `BF1CVTL` and `BF1CVTL2` is selected by FPMR.F8S1. The 8-bit floating-point encoding format for `BF2CVTL` and `BF2CVTL2` is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8)

31	30	29	28				24	23	22	21				17	16				12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	x	1	0	0	0	0	1	0	1	1	1	1	0	Rn		Rd	
		U						size							opcode										

Encoding for the BF1CVTL{2} variant

Applies when `(size == 10)`

```
BF1CVTL{2}  <Vd>.8H, <Vn>.<Ta>
```

Encoding for the BF2CVTL{2} variant

Applies when `(size == 11)`

```
BF2CVTL{2}  <Vd>.8H, <Vn>.<Ta>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP8) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer part = UInt(Q);
constant integer elements = 64 DIV 8;
constant boolean issrc2 = size == '11';
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

Operation

```
CheckFPMREnabled(); AArch64.CheckFPAdvSIMDEnabled();
constant bits(64) operand = Vpart[n, part, 64];
bits(128) result;

for e = 0 to elements-1
    Elem[result, e, 16] = FP8ConvertBF(Elem[operand, e, 8], issrc2, FPCR, FPMR);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.14 BFCVT

Single-precision convert to BFloat16 (scalar)

This instruction converts the single-precision floating-point value in the 32-bit SIMD&FP source register to BFloat16 format and writes the result in the 16-bit SIMD&FP destination register.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Single-precision to BFloat16

(FEAT_BF16)

Encoding

```
BFCVT  <Hd>, <Sn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPEnabled();

constant bits(32) operand = V[n, 32];
constant boolean merge = IsMerging(FPCR);
bits(128) result = if merge then V[d, 128] else Zeros(128);

Elem[result, 0, 16] = FPConvertBF(operand, FPCR);
V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.15 BFCVTN, BFCVTN2

Single-precision convert to BFloat16 (vector)

This instruction reads each single-precision element in the SIMD&FP source vector, converts each value to BFloat16 format, and writes the results in the lower or upper half of the SIMD&FP destination vector. The result elements are half the width of the source elements.

The `BFCVTN` instruction writes the half-width results to the lower half of the destination vector and clears the upper half to zero. The `BFCVTN2` instruction writes the results to the upper half of the destination vector without affecting the other bits in the register.

Vector single-precision to BFloat16

(FEAT_BF16)

Encoding

```
BFCVTN{2}  <Vd>.<Ta>, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer part = UInt(Q);
constant integer elements = 64 DIV 16;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<Ta>
0	4H
1	8H

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(128) operand = V[n, 128];
bits(64) result;

for e = 0 to elements-1
    Elem[result, e, 16] = FPConvertBF(Elem[operand, e, 32], FPCR);

Vpart[d, part, 64] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.16 BFDOT (by element)

BFloat16 dot product to single-precision (vector, by element)

This instruction delimits the source vectors into pairs of BFloat16 elements. The BFloat16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Advanced SIMD

(FEAT_BF16)

Encoding

```
BFDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF);
constant integer n = UInt(Rn);
constant integer m = UInt(M:Rm);
constant integer d = UInt(Rd);
constant integer i = UInt(H:L);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV 32;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	4H
1	8H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the “H:L” fields.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(128)      operand2 = V[m, 128];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    constant bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    constant bits(16) elt2_a = Elem[operand2, 2 * i + 0, 16];
    constant bits(16) elt2_b = Elem[operand2, 2 * i + 1, 16];

    constant bits(32) sum = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR);

V[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.17 BFDOT (vector)

BFloat16 dot product to single-precision (vector)

This instruction delimits the source vectors into pairs of BFloat16 elements.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Advanced SIMD

(FEAT_BF16)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0 1	0	Rm	1	1 1 1 1	1	Rn	Rd
		U				size				opcode			

Encoding

```
BFDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV 32;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	4H
1	8H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    constant bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    constant bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    constant bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];

    constant bits(32) sum = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR);

V[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.18 BFMLALB, BFMLALT (by element)

BFloat16 multiply-add to single-precision (by element)

This instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector

(FEAT_BF16)

Encoding

```
BFMLAL<bt>  <Vd>.4S, <Vn>.8H, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF);
constant integer n = UInt(Rn);
constant integer m = UInt('0':Rm);
constant integer d = UInt(Rd);
constant integer index = UInt(H:L:M);

constant integer elements = 128 DIV 32;
constant integer sel = UInt(Q);
```

Assembler Symbols

<bt>

Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “H:L:M” fields.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(128) result;
constant bits(128) operand1 = V[n, 128];
constant bits(128) operand2 = V[m, 128];
constant bits(128) operand3 = V[d, 128];
constant bits(16) element2  = Elem[operand2, index, 16];

for e = 0 to elements-1
    constant bits(16) element1   = Elem[operand1, 2 * e + sel, 16];
    constant bits(32) addend     = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(addend, element1, element2, FPCR);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.19 BFMLALB, BFMLALT (vector)

BFloat16 multiply-add to single-precision (vector)

This instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector

(FEAT_BF16)

Encoding

```
BFMLAL<bt>  <Vd>.4S, <Vn>.8H, <Vm>.8H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer elements = 128 DIV 32;
constant integer sel = UInt(Q);
```

Assembler Symbols

<bt>

Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(128) operand1 = V[n, 128];
constant bits(128) operand2 = V[m, 128];
constant bits(128) operand3 = V[d, 128];
bits(128) result;

for e = 0 to elements-1
    constant bits(16) element1 = Elem[operand1, 2 * e + sel, 16];
    constant bits(16) element2 = Elem[operand2, 2 * e + sel, 16];
    constant bits(32) addend   = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(addend, element1, element2, FPCR);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.20 BFMMLA (widening)

BFloat16 matrix multiply-accumulate to single-precision

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs two unfused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2x4 matrix of BFloat16 values in the first source vector with the 4x2 matrix of BFloat16 values in the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs two fused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2x4 matrix of BFloat16 values in the first source vector with the 4x2 matrix of BFloat16 values in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way fused dot products per destination element.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Advanced SIMD

(FEAT_BF16)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	1	1	0	1 1 1	0	0 1	0	Rm	1	1 1 0 1	1	Rn	Rd
	Q	U				size				opcode			

Encoding

```
BFMMLA  <Vd>.4S, <Vn>.8H, <Vm>.8H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(128) op1 = V[n, 128];
constant bits(128) op2 = V[m, 128];
constant bits(128) acc = V[d, 128];

V[d, 128] = BFMatMulAddH(acc, op1, op2, FPCR);
```

Operational Information

Arm expects that the BFMMLA instruction will deliver a peak BFloat16 multiply throughput that is at least as high as can be achieved using two BFDOT (vector) instructions, with a goal that it should have significantly higher throughput.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.21 BIC (vector, immediate)

Bitwise bit clear (vector, immediate)

This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Shifted immediate

(FEAT_AdvSIMD)

31	30	29	28									19	18	17	16	15			12	11	10	9	8	7	6	5	4	0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd	
		op														cmode				o2								

Encoding for the 16-bit variant

Applies when `(cmode == 10x1)`

```
BIC  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit variant

Applies when `(cmode == 0xx1)`

```
BIC  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer rd = UInt(Rd);
constant integer datasize = 64 << UInt(Q);
constant bits(64) imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
constant bits(datasize) imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP register, encoded in the “Rd” field.

<T>

For the “16-bit” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1	8H

For the “32-bit” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8>

Is an 8-bit immediate encoded in “a:b:c:d:e:f:g:h”.

<amount>

For the “16-bit” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit” variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[rd, datasize];
V[rd, datasize] = operand AND NOT(imm);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.22 BIC (vector, register)

Bitwise bit clear (vector, register)

This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0 1	1	Rm	0 0 0 1 1	1	Rn	Rd
		U				size			opcode			

Encoding

```
BIC  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = NOT(V[m, datasize]);
V[d, datasize] = operand1 AND operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.23 BIF

Bitwise insert if false

This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

Encoding

```
BIF  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[d, datasize];
constant bits(datasize) operand2 = NOT(V[m, datasize]);
constant bits(datasize) operand3 = V[n, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand3) AND operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.24 BIT

Bitwise insert if true

This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

Encoding

```
BIT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[d, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[n, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand3) AND operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.25 BSL

Bitwise select

This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

Encoding

```
BSL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[m, datasize];
constant bits(datasize) operand2 = V[d, datasize];
constant bits(datasize) operand3 = V[n, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand3) AND operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.26 CLS (vector)

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&FP register, places the result into a vector, and writes the vector to the destination SIMD&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.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 0 0 0 0	0 0 1 0 0	1 0	Rn	Rd
		U						opcode			

Encoding

```
CLS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant integer count = CountLeadingSignBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.27 CLZ (vector)

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&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

Encoding

```
CLZ  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant integer count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.28 CMEQ (register)

Compare bitwise equal (vector)

This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1	1	1	0	1	1	1	Rm	1	0	0	0	1	1	Rn	Rd
		U						size				opcode							

Encoding

```
CMEQ  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	0	size		1	Rm	1	0	0	0	1	1	Rn	Rd
		U										opcode							

Encoding

```
CMEQ  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = if element1 == element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.29 CMEQ (zero)

Compare bitwise equal to zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

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

Encoding

```
CMEQ  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

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

Encoding

```
CMEQ  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element == 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.30 CMGE (register)

Compare signed greater than or equal (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&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&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	1	0	1	1 1 1	0	1 1	1	Rm	0 0 1 1	1	1	Rn	Rd
		U				size				eq			

Encoding

```
CMGE  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0 1 1	1	1	Rn	Rd
		U								eq			

Encoding

```
CMGE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer element2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = if element1 >= element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.31 CMGE (zero)

Compare signed greater than or equal to zero (vector)

This instruction reads each vector element in the source SIMD&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&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	1 0 0 0 0	0	1 0 0	0	1	0	Rn		Rd	
		U						size					op						

Encoding

```
CMGE  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	size	1	0 0 0 0	0	1 0 0	0	1	0	Rn		Rd	
		U											op						

Encoding

```
CMGE  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element >= 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.32 CMGT (register)

Compare signed greater than (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&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&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	1	0	1	1 1 1	0	1 1	1	Rm	0 0 1 1	0	1	Rn	Rd
		U				size				eq			

Encoding

```
CMGT  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0 1 1	0	1	Rn	Rd
		U								eq			

Encoding

```
CMGT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer element2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = if element1 > element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.33 CMGT (zero)

Compare signed greater than zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

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

Encoding

```
CMGT  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

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

Encoding

```
CMGT  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element > 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.34 CMHI (register)

Compare unsigned higher (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&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&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1 1	1	Rm	0 0 1 1	0	1	Rn	Rd
		U				size				eq			

Encoding

```
CMHI  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0 1 1	0	1	Rn	Rd
		U								eq			

Encoding

```
CMHI  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer element2;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = if element1 > element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.35 CMHS (register)

Compare unsigned higher or same (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&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&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1 1	1	Rm	0 0 1 1	1	1	Rn	Rd
		U				size				eq			

Encoding

```
CMHS  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0 1 1	1	1	Rn	Rd
		U								eq			

Encoding

```
CMHS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer element2;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = if element1 >= element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.36 CMLE (zero)

Compare signed less than or equal to zero (vector)

This instruction reads each vector element in the source SIMD&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&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	1 0 0 0 0	0	1 0 0	1	1	0	Rn		Rd	
		U						size					op						

Encoding

```
CMLE  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	size	1	0 0 0 0	0	1 0 0	1	1	0	Rn		Rd	
		U											op						

Encoding

```
CMLE  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element <= 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.37 CMLT (zero)

Compare signed less than zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

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

Encoding

```
CMLT  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

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

Encoding

```
CMLT  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element < 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.38 CMTST

Compare bitwise test bits nonzero (vector)

This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0	1	0	1	1 1 1	0	1 1	1	Rm	1 0 0 0 1	1	Rn	Rd
		U				size			opcode			

Encoding

```
CMTST  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0 0 0 1	1	Rn	Rd
		U							opcode			

Encoding

```
CMTST  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = if !IsZero(element1 AND element2) then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.39 CNT

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&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

Encoding

```
CNT  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant integer count = BitCount(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.40 DUP (element)

Duplicate vector element to vector or scalar

This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar).

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	0	0	0	0	imm5	0	0	0	0	0	1	Rn	Rd
		op														imm4			

Encoding

```
DUP  <V><d>, <Vn>.<T>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer index = UInt(imm5<4:size+1>);
constant integer idxdsize = 64 << UInt(imm5<4>);
constant integer esize = 8 << size;
constant integer datasize = esize;
constant integer elements = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	0	0	0	imm5	0	0	0	0	0	1	Rn	Rd
		op														imm4			

Encoding

```
DUP  <Vd>.<T>, <Vn>.<Ts>[<index>]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
if imm5 == 'x1000' && Q == '0' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer index = UInt(imm5<4:size+1>);
constant integer idxdsize = 64 << UInt(imm5<4>);
constant integer esize = 8 << size;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T>

For the “Scalar” variant: is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

For the “Vector” variant: is an arrangement specifier, encoded in “imm5:Q”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<index>

Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ts>

Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

Alias Conditions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (scalar)	Unconditionally

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(idxdsize) operand = V[n, idxdsize];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.41 DUP (general)

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&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.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 21	20 16	15	14 11	10	9 5	4 0
0	Q	0	0	1 1 1	0 0	0 0	imm5	0	0 0 0 1	1	Rn	Rd
		op							imm4			

Encoding

```
DUP  <Vd>.<T>, <R><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
if imm5 == 'x1000' && Q == '0' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
// imm5<4:size+1> is IGNORED
constant integer esize = 8 << size;
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “imm5:Q”:

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	Q	<T>
x1000	0	RESERVED
x1000	1	2D

<R>

Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

Unspecified bits in “imm5” are ignored but should be set to zero by an assembler.

<n>

Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) element = X[n, esize];
constant integer elements = datasize DIV esize;
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.42 EOR3

Three-way exclusive-OR

This instruction performs a three-way exclusive-OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA3)

31 28	27 25	24 23	22 21	20 16	15	14 10	9 5	4 0
1 1 0 0	1 1 1	0 0	0 0	Rm	0	Ra	Rn	Rd
			Op0					

Encoding

```
EOR3  <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Va>

Is the name of the third SIMD&FP source register, encoded in the “Ra” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[m, 128];
constant bits(128) operand2 = V[n, 128];
constant bits(128) operand3 = V[a, 128];
V[d, 128] = operand2 EOR operand1 EOR operand3;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.43 EOR (vector)

Bitwise exclusive-OR (vector)

This instruction performs a bitwise exclusive-OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

Encoding

```
EOR  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[m, datasize];
constant bits(datasize) operand2 = Zeros(datasize);
constant bits(datasize) operand3 = Ones(datasize);
constant bits(datasize) operand4 = V[n, datasize];
V[d, datasize] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.44 EXT

Extract vector from pair of vectors

This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

Figure C7-1 EXT doubleword operation for Q = 0 and imm4<2:0> = 3

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29 28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	1 0	1 1 1	0	0 0	0	Rm	0	imm4	0	Rn	Rd
					op2							

Encoding

```
EXT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if Q == '0' && imm4<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
constant integer position = 8 * UInt(imm4);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<index>

Is the lowest numbered byte element to be extracted, encoded in “Q:imm4”:

Q	imm4<3>	<index>
0	0	UInt(imm4<2:0>)
0	1	RESERVED
1	x	UInt(imm4)

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) hi = V[m, datasize];
constant bits(datasize) lo = V[n, datasize];
constant bits(datasize*2) concat = hi : lo;

V[d, datasize] = concat<(position+datasize)-1:position>;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.45 F1CVTL, F1CVTL2, F2CVTL, F2CVTL2

8-bit floating-point convert to half-precision (vector)

This instruction converts each 8-bit floating-point element from the lower or upper half of the source vector to half-precision while downscaling the value, and places the results in the 16-bit elements of the destination vector. `F1CVTL` and `F2CVTL` convert the elements from the lower half of the source vector while scaling the values by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ and $2^{-\text{UInt(FPMR.LSCALE2[3:0])}}$, respectively. `F1CVTL2` and `F2CVTL2` convert the elements from the upper half of the source vector while scaling the values by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ and $2^{-\text{UInt(FPMR.LSCALE2[3:0])}}$, respectively.

The 8-bit floating-point encoding format for `F1CVTL` and `F1CVTL2` is selected by FPMR.F8S1. The 8-bit floating-point encoding format for `F2CVTL` and `F2CVTL2` is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8)

Encoding for the F1CVTL{2} variant

Applies when `(size == 00)`

```
F1CVTL{2}  <Vd>.8H, <Vn>.<Ta>
```

Encoding for the F2CVTL{2} variant

Applies when `(size == 01)`

```
F2CVTL{2}  <Vd>.8H, <Vn>.<Ta>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP8) then EndOfDecode(Decode_UNDEF);
constant integer n = UInt(Rn);
constant integer d = UInt(Rd);
constant integer part = UInt(Q);
constant integer elements = 64 DIV 8;
constant boolean issrc2 = size == '01';
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

Operation

```
CheckFPMREnabled(); AArch64.CheckFPAdvSIMDEnabled();
constant bits(64) operand = Vpart[n, part, 64];
bits(128) result;

for e = 0 to elements-1
    Elem[result, e, 16] = FP8ConvertFP(Elem[operand, e, 8], issrc2, FPCR, FPMR);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.46 FABD

Floating-point absolute difference (vector)

This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1	1	1	0	1	1	0	Rm	0	0	0	1	0	1	Rn	Rd
		U						a						opcode					

Encoding

```
FABD  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1	1	1	0	1	sz	1	Rm	1	1	0	1	0	1	Rn	Rd
		U												opcode					

Encoding

```
FABD  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 0	Rm	0 0	0 1 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FABD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1	Rm	1 1 0 1 0	1	Rn	Rd
		U								opcode			

Encoding

```
FABD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
constant boolean merge = elements == 1 && IsMerging(FPCR);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    constant bits(esize) diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = FPAbs(diff, FPCR);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.47 FABS (vector)

Floating-point absolute value (vector)

This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	1	1 1 1 1	0 0	0 1 1 1 1	1 0	Rn	Rd
		U				a			opcode			

Encoding

```
FABS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	1	sz	1 0 0 0 0	0 1 1 1 1	1 0	Rn	Rd
		U							opcode			

Encoding

```
FABS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPAbs(element, FPCR);

V[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.48 FABS (scalar)

Floating-point absolute value (scalar)

This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FABS  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FABS  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FABS  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPEnabled();

constant bits(esize) operand = V[n, esize];
bits(128) result = if IsMerging(FPCR) then V[d, 128] else Zeros(128);

Elem[result, 0, esize] = FPAbs(operand, FPCR);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.49 FACGE

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&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&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&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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 exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	0	1	0	Rm	0	0	1	0	1	1	Rn	Rd
		U				E								ac			

Encoding

```
FACGE  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	0	sz	1	Rm	1 1 1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGE  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 0	Rm	0 0	1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1	Rm	1 1 1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
constant boolean merge = elements == 1 && IsMerging(FPCR);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    element1 = FPAbs(element1, FPCR);
    element2 = FPAbs(element2, FPCR);
    test_passed = FPCompareGE(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.50 FACGT

Floating-point absolute compare greater than (vector)

This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&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&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&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 exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1	1	0	Rm	0	0	1	0	1	1	Rn	Rd
		U				E								ac			

Encoding

```
FACGT  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1	sz	1	Rm	1 1 1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGT  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 0	Rm	0 0	1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1	Rm	1 1 1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
constant boolean merge = elements == 1 && IsMerging(FPCR);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    element1 = FPAbs(element1, FPCR);
    element2 = FPAbs(element2, FPCR);
    test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.51 FADD (vector)

Floating-point add (vector)

This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

Encoding

```
FADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

Encoding

```
FADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
V[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.52 FADD (scalar)

Floating-point add (scalar)

This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20	16	15	13	12	11 10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1	Rm		0 0 1		0	1 0	Rn		Rd	

M S op

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FADD  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FADD  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FADD  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

bits(128) result = if IsMerging(FPCR) then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPAdd(operand1, operand2, FPCR);

V[d, 128] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.53 FADDP (scalar)

Floating-point add pair of elements (scalar)

This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	1	1	0	Rn	Rd
		U							sz								opcode						

Encoding

```
FADDP  H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 32;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	1	1	0	Rn	Rd
		U															opcode						

Encoding

```
FADDP  <V><d>, <Vn>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize * 2;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<T>

Is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
constant bits(esize) element1 = Elem[operand, 0, esize];
constant bits(esize) element2 = Elem[operand, 1, esize];
V[d, esize] = FPAdd(element1, element2, FPCR);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.54 FADDP (vector)

Floating-point add pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&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&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 exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

Encoding

```
FADDP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

Encoding

```
FADDP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

    V[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.55 FAMAX

Floating-point absolute maximum

This instruction determines the maximum absolute value from floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and places the results in the corresponding elements of the destination vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is the Default NaN, with the sign bit set to 0.
- Denormalized inputs and results are never flushed to zero, as if FPCR.{FZ, FZ16, FIZ} are all 0.
- Denormalized inputs never generate an Input Denormal floating-point exception.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FAMINMAX)

Encoding

```
FAMAX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = if Q == '1' then 128 else 64;
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD && FEAT_FAMINMAX)

Encoding

```
FAMAX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = if Q == '1' then 128 else 64;
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “size<0>:Q”:

size<0>	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant bits(esize) op1 = Elem[operand1, e, esize];
    constant bits(esize) op2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAbsMax(op1, op2, FPCR);
V[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.56 FAMIN

Floating-point absolute minimum

This instruction determines the minimum absolute value from floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and places the results in the corresponding elements of the destination vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is the Default NaN, with the sign bit set to 0.
- Denormalized inputs and results are never flushed to zero, as if FPCR.{FZ, FZ16, FIZ} are all 0.
- Denormalized inputs never generate an Input Denormal floating-point exception.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FAMINMAX)

Encoding

```
FAMIN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = if Q == '1' then 128 else 64;
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD && FEAT_FAMINMAX)

Encoding

```
FAMIN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = if Q == '1' then 128 else 64;
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “size<0>:Q”:

size<0>	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant bits(esize) op1 = Elem[operand1, e, esize];
    constant bits(esize) op2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAbsMin(op1, op2, FPCR);
V[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.57 FCADD

Floating-point complex add

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source 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 exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 13	12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	0	Rm	1	1 1	rot	0	1	Rn	Rd

U

Encoding

```
FCADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then EndOfDecode(Decode_UNDEF);
if size == '00' then EndOfDecode(Decode_UNDEF);
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<rotate>

Is the rotation, encoded in “rot”:

rot	<rotate>
0	90
1	270

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '0'
            element1 = FPNeg(Elem[operand2, e*2+1, esize], FPCR);
            element3 = Elem[operand2, e*2, esize];
        when '1'
            element1 = Elem[operand2, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize], FPCR);
    Elem[result, e*2,   esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR);
    Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR);

V[d, datasize] = result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.58 FCCMP

Floating-point conditional quiet compare (scalar)

This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass, then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

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 exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28 … 24	23 22	21	20 … 16	15 … 12	11 10	9 … 5	4	3 … 0
0	0	0	1 1 1 1 0	ftype	1	Rm	cond	0 1	Rn	0	nzcv
M		S								op	

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FCCMP  <Hn>, <Hm>, #<nzcv>, <cond>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FCCMP  <Sn>, <Sm>, #<nzcv>, <cond>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FCCMP  <Dn>, <Dm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 8 << UInt(ftype EOR '10');
constant bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPEnabled();

constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

if ConditionHolds(condition) then
    constant boolean signal_all_nans = FALSE;
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
PSTATE.<N,Z,C,V> = flags;
```

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential